目錄

Gradle-2-User-Guide

Chinese translation of Gradle 2 User Guide . The latest release of Gradle is version 2.14, released on 14th March 2016.You can also see the demos of the guide here.There is a GitBook version of the book: http://waylau.gitbooks.io/gradle-2-user-guide/. Let's READ!

中文翻译《Gradle 2 用户指南》。Gradle 最新版本是 2.14 （发布于 2016-3-14）。利用业余时间对此进行翻译，并在原文的基础上，插入配图，图文并茂方便用户理解，并提供丰富的示例源码。如有勘误欢迎指正。如有兴趣，也可以参与到本翻译工作中来 :)

如果你想尝试使用 Gradle 3 的新特性，也可以参考笔者的另外翻译的一本开源书《Gradle 3 用户指南》。

Get Started 如何开始阅读

选择下面入口之一：

- https://github.com/waylau/Gradle-2-User-Guide/ 的 SUMMARY.md（源码）
- http://waylau.gitbooks.io/gradle-2-user-guide/ 点击 Read 按钮（同步更新，国内访问速度一般）

Code 源码

书中所有示例源码，移步至https://github.com/waylau/Gradle-2-User-Guide-Demos

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/Gradle-2-User-Guide/issues

Contact 联系作者:

- Blog: waylau.com
- Gmail: waylau521(at)gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

Chapter 1. Introduction 介绍

Gradle 为Java（JVM）世界提供快速构建的工具。提供如下功能：

- 一个非常灵活的通用构建工具,如 Ant
- 方便从 Maven 中切换过来。但我们从不强制
- 对多项目构建具有强有力的支持
- 很强的依赖性管理（基于 Apache Ivy）
- 对你现有的 Maven 或者 Ivy 库全力支持
- 支持传递依赖管理，不需要远程仓库或者 pom.xml 和 ivy.xml 文件
- Ant 的任务和构建是一等公民
- Groovy 为构建使用脚本
- 一个丰富的域模型描述你的构建

在Chapter 2. Overview 总览 可以看到更多 Gradle 的概况。后面也有更多教程再等着你。

1.1. About this user guide 关于本用户指南

本指南如同 Gradle 一样不断的再更新，如有勘误或者改正之处，请给本指南更多建议，可以访问Gradle web site

译者注：对于本翻译如有勘误或者改正之处，点此提问。

在用户指南，你会发现一些图表示 Gradle 任务之间的依赖关系。这些使用一些类似 UML 依赖符号，使箭从一个任务到任务（第一个任务所依赖的）。

Chapter 2. Overview 总览

2.1. Features 特性

下面列出了一些 Gradle 的特性：

Declarative builds and build-by-convention声明式构建，符合公约

gradle 的核心是在 基于 Groovy 对 Domain Specific Language (DSL)语言进行一个丰富的扩展。根据喜好，Gradle 将陈述建立下一级提供声明性语言元素。这些元素也提供支持 Java，Groovy，OSGi，Web和Scala 项目。甚至更多，这说明语言是可扩展的。添加您自己的新语言元素或加强现有的，从而提供了简洁，易于维护和易于理解的构建

Language for dependency based programming 依赖型编程语言

声明式语言位于一个通用的任务图，你可以充分利用你的构建。它提供了适应您的独特需求的最大灵活性的工具。

Structure your build 良好的结构

工具的柔软性和丰富性允许您用一般性设计原则来构建项目。你可以创建一个结构良好，易于维护，易于理解的建立。

Deep API 深层次的API

工具允许您监视和自定义配置和执行行为

Gradle scales 可伸缩

Gradle 伸缩性能非常好。它会增加你的生产力，从简单的单项目到建立庞大的企业多项目建设。

Multi-project builds 多项目构建

Gradle 的多项目构建支持是非常优秀的。项目依赖性是一等公民。我们允许你在多项目构建的项目的关系中建模，即使他们是问题域。Gradle 会遵从你的布局而不是相反。

Gradle 提供部分的基础之上。如果你建立一个子项目 Gradle 负责构建所有子项目依赖于子项目。您也可以选择重建依赖于特定的子项目的子项目。再加上增量构建，对于较大的构建这很节省很多时间，。

Many ways to manage your dependencies 多途径管理依赖

不同的团队喜欢不同的方式来管理他们的外部依赖。Gradle 提供任何便捷的策略支持。从远程 Maven 和 Ivy 库的依赖管理与到本地文件系统上 jars 或目录。

Gradle is the first build integration tool Gradle是第一个构建集成工具

Ant 任务是一等公民。更有趣的是，Ant 的项目也都是一等公民。Gradle 提供深入的引用给 Ant 项目，在运行时，可以转换 Ant 目标到 原生的Gradle 任务。你可以依靠他们的工具，可以提高他们的工具，你甚至可以在build.xml 宣布对 Gradle 任务的依赖。相同的集成提供了性能，路径，等...

Gradle 支持现有的 Maven 或 Ivy 仓库依赖关系。工具还提供了一个转换器将 Maven pom.xml 转成 Gradle 脚本。Maven 项目运行的进口就快来了。

Ease of migration 易迁移

Gradle 可以适应任何已有的结构。我们通常建议写测试，确保与生产环境类似。这样的迁移是更少的破坏性和尽可能的可靠。这是继重构应用小步骤的最佳实践。

Groovy 语言

工具的构建脚本是用 Groovy，不是XML。但是，不像其他的方法，这不是简单地将动态语言的原始脚本进行能力的扩展。这只会导致一个保持非常困难的构建。工具的总体设计是面向的是将 Gradle 作为一种语言，而不是一个严格的框架。工具提供了一些标准的故事，但他们不做任何形式的限制。这是我们的一个主要特点。

The Gradle wrapper 关于Gradle的包装

该Gradle包装允许你机器上没有安装Gradle工具也能执行 Gradle 的构建

Free and open source 免费开源

遵守ASL开源协议

2.2. Why Groovy? 为啥用 Groovy

我们认为，当使用构建脚本作为内部 DSL （基于动态语言）比 XML 有更大的优势。有很多动态语言，但为啥是 Groovy？答案是在于上下文工具的操作。虽然 Gradle 是一个通用的构建工具，这是它的核心，但它的主要焦点还是是 Java 项目。在这样的项目中，团队成员更加熟悉 Java。我们考虑的是编译应该都所有成员来说是尽可能的透明。

你可能会说，为什么不使用 Java 作为构建脚本。这里有一个问题，就是对于团队的最高的透明度和最低的学习曲线，但是 由于 Java 语言的限制，作为构建语言效果并不理想（参考 http://www.defmacro.org/ramblings/lisp.html)可以看到 Ant, XML, Java 和 Lisp 的对比，有趣的是，Java 的语法实际上是 Groovy 的语法。）。其他语言， Python, Groovy 或者 Ruby 都更能胜任这个工作。我们选择 Groovy 是因为对于 Java 使用者来说有更高的透明度。它的基本语法与 Java 类似，包括 本文系统，包结构和其他方面。Groovy 提供了最重要内容但都是符合 Java 基础功能的。

对于对 Python 或 Ruby 知识拥有强烈的学习欲望的 Java 开发者来说，上述论点不适用。该工具的设计非常适合于创建另一个建立在 JRuby 和Jython 脚本引擎。对于我们来说暂时它只是不具有最高优先级。我们高兴地支持任何社区的努力来创建额外的构建脚本引擎。

Chapter 3. Tutorials 教程

3.1. Getting Started 开始

下面教程将介绍基本的 Gradle 知识。

Chapter 4. Installing Gradle 安装

描述如何安装

Chapter 6. Build Script Basics 构建脚本的基础知识识

介绍基本的构建脚本元素：项目和任务

Chapter 7. Java Quickstart 快速开始 Java

展示如何开始使用 Gradle 的基于约定的构建 来支持 Java 项目

Chapter 8. Dependency Management Basics 依赖管理的基础知识

显示如何使用 Gradle 的依赖管理。

Chapter 9. Groovy Quickstart 快速开始 Groovy

使用 Gradle 的基于约定的构建 来支持 Groovy 项目

Chapter 10. Web Application Quickstart 快速开始 Web 应用

使用 Gradle 的基于约定的构建 来支持 web 项目

Chapter 4. Installing Gradle 安装

4.1. Prerequisites 前置条件

Gradle 需要 Java JDK 或者 JRE，版本是 6 及以上。Gradle 将会装载自己的 Groovy 库，因此，Groovy 不需要被安装。任何存在的 Groovy 安装都会被 Gradle 忽略。

Gradle 使用你 path 中的 JDK,或者，您可以设置 java_home 环境变量来指向所需的 JDK 安装目录。

4.2. Download 下载

下载 Gradle 的发布包.

4.3. Unpacking 解压

Gradle 的发布包被打包成 ZIP。完整的发布包含:

- Gradle 二进制
- 用户指南 (HTML 和 PDF)
- DSL 参考指南
- API 文档(Javadoc 和 Groovydoc)
- 扩展示例，包括用户指南中引用的例子，以及一些完整的和更复杂的构建可以作为自己开始的构建。
- 二进制源文件。这是只供参考。如果你想编译 Gradle 你需要下载源发布包或从源库资源中检出。请参阅网站

4.4. Environment variables 环境变量

添加 `GRADLE_HOME/bin` 到你的 path 环境变量

4.5. Running and testing your installation 运行和测试安装

通过 `gradle` 命令运行 Gradle。 `gradle -v` 用来查看安装是否成功。输出 Gradle 的版本信息和本地变量配置信息 (Groovy, JVM version, OS, 等)。Gradle 显示的版本信息应该和你下载的匹配

4.6. JVM options 虚拟机选项

虚拟机选项可以设置 Gradle 的运行环境变量。可以使用 GRADLE_OPTS 或者 JAVA_OPTS，或者两个都选。 JAVA_OPTS 约定和 java 共享环境变量。典型的案例是在 JAVA_OPTS 设置 HTTP 代理，在 GRADLE_OPTS 设置内存。这些变量也可在`gradle` 或者 `gradlew` 脚本开始时设置

注意：目前无法在命令行工具为 Gradle 设置 JVM 选项

Chapter 5. Troubleshooting 问题解决

本章未完，还在进行中

当你在使用 Gradle 途中有任何问题，请看本章如何解决

5.1. Working through problems 处理问题

当你遇到问题，首先是更新到最新版本。最新版一般是修复了 bug 和添加了新特性。

如果您使用的是 Gradle Daemon 守护进程，尝试暂时禁用该守护进程（您可以通过命令行 --no-daemon）。有关故障排除的守护进程的更多信息，位于Chapter 19. The Gradle Daemon 守护进程

5.2. Getting help 获取帮助

在线论坛http://forums.gradle.org. 可以提问或者建议。

如果有任何问题，发布一个问题到论坛报告是获得帮助最快的方法。这也是后期的改进建议或新思想的地方。开发团队经常通过论坛发布新项目和消息宣布，这是让它 跟 Gralde 的开发保持最新的一个伟大的方式。

Chapter 6. Build Script Basics 构建脚本的基本

在 Gradle 中两个顶级概念：project（项目）和 task 任务）

所有 Gradle 都有一个或多个 project 构成。project 的展现取决于 Gradle 所做的工作。举例。project 可以是一个 JAR 库 或者是 web 应用。它可以是由项目生产 JAR 组成发布的 ZIP。一个 project 不一定 代表一个东西要构建。它可能是一件要做的事，如将应用程序部署到工作台或生产环境。如果这看起来有点模糊，现在不要担心。Gradle 基于约定的构建支持增加一个更具体的定义的 project。

每个项目都是由一个或多个 task。一个 task 代表了一个构建生成的原子的作品。这可能是编写一些类，创建一个 JAR ，生成 Javadoc，或发布一些库。

现在，我们将看看在构建一个 project 时定义一些简单的 task 。后面的章节将介绍多个 project 和更多的 task 。

6.2. Hello world

运行 Gradle 是使用 gradle 命令行。命令行会寻找项目的根目录下 build.gradle 的文件（有关命令行，详见 Appendix D. Gradle Command Line 命令行），这个就是构建的脚本，或者严格说是构建的配置脚本。他定义了project（项目）和 task 任务）。

尝试输出，创建一个 `build.gradle` 命名的文件：

Example 6.1. Your first build script

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

命令行切换到包含 build.gradle 文件的目录，执行 `gradle -q hello`

Example 6.2. Execution of a build script

输出为：

```
> gradle -q hello
Hello world!
```

这个脚本定义了一个名字是 `hello` 的 task,并且添加了动作。当运行 `gradle hello` ,Gradle 执行这个 `hello` task，接着执行里面的动作。这里的动作只是简单的包含了一些可以执行的 Groovy 代码。

看上去很像 Ant ，不错，Gradle task 是相当于 Ant 的 target，但是你将看到，他们更强大。我们使用了跟 Ant 不同的术语，因为 task 比 target 更富表现力。

不幸的是，这一术语与 Ant 有冲突，Ant 调用它的命令行，如 javac 或copy 称之为 task 。所以当我们谈论的 task ，默认说的是 Gradle task ，这是相当 Ant 的 target。如果我们谈论的 Ant 的 task （Ant 命令），我们明确地说的 Ant task。

命令行加中 `-q` 的作用

q 是 quiet 的简写，意思是要安静、干净的输出。如果不加 `-q` 则会输出日志。详见Chapter 18. Logging 日志.下面是对比

6.3. A shortcut task definition 快捷 task 定义

定义 task 可以使用快捷方式，这样更简明。

Example 6.3. A task definition shortcut

build.gradle

```
task hello << {
    println 'Hello world!'
}
```

再次执行，得到相同的输出。在下面的文章中，我们都会采用这种定义方式。

6.4. Build scripts are code 构建的脚本都是代码

工具的构建脚本给你完整的 Groovy 的功能。作为开胃菜，看看这个：

Example 6.4. Using Groovy in Gradle's tasks

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

执行 `gradle -q upper` 输出

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

或者

Example 6.5. Using Groovy in Gradle's tasks

build.gradle

```
task count << {
    4.times { print "$it " }
}
```

执行 `gradle -q count` 输出

```
> gradle -q count
0 1 2 3
```

6.5. Task dependencies 依赖

可以声明 task 与 其他 task 的依赖

Example 6.6. Declaration of task that depends on other task

build.gradle

```
task hello << {
    println 'Hello world!'
}
task intro(dependsOn: hello) << {
    println "I'm Gradle"
}
```

执行 `gradle -q intro` 输出

```
> gradle -q intro
Hello world!
I'm Gradle
```

添加一个依赖，相应的 task 不需要存在

Example 6.7. Lazy dependsOn - the other task does not exist (yet)

build.gradle

```
task taskX(dependsOn: 'taskY') << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
```

执行 `gradle -q taskX` 输出

```
> gradle -q taskX
taskY
taskX
```

taskX 的依赖 taskY 是在 taskY 定义之前声明的。这个在多 project 构建时很重要。关于 task 的依赖详见 Chapter 15. More about Tasks 更多关于任务

请注意不要使用快捷符号，当引用的 task 还没有定义的情况下。

6.6. Dynamic tasks 动态 task

Groovy 的能力不仅仅是定义一个 task。例如，你也可以用它来动态创建的 task。

Example 6.8. Dynamic creation of a task

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
```

执行 `gradle -q task1` 输出

```
> gradle -q task1
I'm task number 1
```

6.7. Manipulating existing tasks 利用现有的任务

一旦 task 创建，他们可以通过一个 API 访问。例如，在运行时您可以使用此动态添加依赖到 task 。Ant 不允许这样的事情。

Example 6.9. Accessing a task via API - adding a dependency

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
task0.dependsOn task2, task3
```

执行 `gradle -q task0` 输出

```
> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

或者 可以添加行为到一个已经存在 task 中

Example 6.10. Accessing a task via API - adding behaviour

build.gradle

```
task hello << {
    println 'Hello Earth'
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello << {
    println 'Hello Jupiter'
}
```

执行 `gradle -q hello` 输出

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

doFirst 和 doLast 可以多次执行调用。他们在开始或结束的 task 动作清单中添加动作。task 执行时，按动作列表的顺序执行的动作。操作符 `<<` 仅仅是 doLast 的别名。

6.8. Shortcut notations 快捷符号

在前面的示例中已经注意到，有一个方便的符号访问现有的 task 。每个 task 可以作为构建脚本的一个属性：

Example 6.11. Accessing task as a property of the build script

build.gradle

```
task hello << {
    println 'Hello world!'
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

执行 `gradle -q hello` 输出

```
> gradle -q hello
Hello world!
Greetings from the hello task.
```

这使得代码可读性增强，尤其是当使用的插件提供的 task ，如 compile task

6.9. Extra task properties 额外 task 属性

可以添加自己属性到 task ,添加 `myProperty` 属性，设置 、 `ext.myProperty` 初始值,从这一点上，该属性可以读取和设置就像一个预定义的任务属性。

Example 6.12. Adding extra properties to a task

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties << {
    println myTask.myProperty
}
```

执行 `gradle -q printTaskProperties` 输出

```
> gradle -q printTaskProperties
myValue
```

task 不对额外属性做限制，更多详见Chapter 13. Writing Build Scripts 编写构建脚本 中 13.4.2 节 “Extra properties”.

6.10. Using Ant Tasks 使用 Ant task

Ant task 是 Gradle 一等公民。 Gradle 给 Ant task 提供了不错的整合通过简单依靠于 Gradle 。Groovy 被奇异的 AntBuilder 装载。从 Gradle 使用 Ant task 比使用 build.xml 文件更方便和更强大。从下面的例子中，你可以学习如何执行 Ant task 和如何访问 Ant 属性：

Example 6.13. Using AntBuilder to execute ant.loadfile target

build.gradle

```
task loadfile << {
    def files = file('antLoadfileResources').listFiles().sort()
    files.each { File file ->
        if (file.isFile()) {
            ant.loadfile(srcFile: file, property: file.name)
            println " *** $file.name ***"
            println "${ant.properties[file.name]}"
        }
    }
}
```

执行 `gradle -q loadfile` 输出

```
> gradle -q loadfile
*** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration  over contract negotiation
Responding to change over following a plan
 *** gradle.manifesto.txt ***
Make the impossible possible, make the possible easy and make the easy elegant.
(inspired by Moshe Feldenkrais)
```

更多关于 构建脚本中使用 Ant ,详见 Chapter 17. Using Ant from Gradle 从 Gradle 使用 Ant

6.11. Using methods 使用方法

Gradle 延伸取决你如何组织的建造逻辑。上面的例子中的第一级别的组织你的构建逻辑，是提取方法。

Example 6.14. Using methods to organize your build logic

build.gradle

```
task checksum << {
    fileList('../antLoadfileResources').each {File file ->
        ant.checksum(file: file, property: "cs_$file.name")
        println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
    }
}

task loadfile << {
    fileList('../antLoadfileResources').each {File file ->
        ant.loadfile(srcFile: file, property: file.name)
        println "I'm fond of $file.name"
    }
}

File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

执行 `gradle -q loadfile` 输出

```
> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

以后你会发现这样的方法可以在多 project 构建的子 project 之间共享。如果你建立逻辑变得越来越复杂，Gradle 为您提供其他工具很方便的方式来组织它。我们有专门一章Chapter 60. Organizing Build Logic

6.12. Default tasks 默认 task

Gradle 允许你定义一个或多个默认 task 给你的构建

Example 6.15. Defining a default tasks

build.gradle

```
defaultTasks 'clean', 'run'

task clean << {
    println 'Default Cleaning!'
}

task run << {
    println 'Default Running!'
}

task other << {
    println "I'm not a default task!"
}
```

执行 `gradle -q` 输出

```
> gradle -q
Default Cleaning!
Default Running!
```

这个等于执行了 `gradle clean run` ，在多 project 中构建所有的子 project 都可以有自己具体的默认 task 。如果 子 project 没有明确的默认 task，则执行父 project 的默认 task（如果定义的话）

6.13. Configure by DAG 通过 DAG 配置

以后会详细描述（见Chapter 56. The Build Lifecycle 构建生命周期 ），Gradle 有配置阶段和执行阶段。配置阶段后，Gradle 知道所有的 task 应该执行。Gradle 提供给你一个钩子来利用这些信息。这个用例将检查发布 的 task 是否是要执行的 task。基于此，你可以赋予不同的值到一些变量。

在下面的例子中，在不同 version 变量中的 distribution 和 release task 执行结果不同。

Example 6.16. Different outcomes of build depending on chosen tasks

build.gradle

```
task distribution << {
    println "We build the zip with version=$version"
}

task release(dependsOn: 'distribution') << {
    println 'We release now'
}

gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(release)) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

执行 `gradle -q distribution` 输出

```
> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
```

执行 `gradle -q release` 输出

```
> gradle -q release
We build the zip with version=1.0
We release now
```

`whenReady` 影响了 release task 在 release task 被执行之前。同样适用于 release task 不是主 task 的情况（比如，task 被 gradle 命令通过了）

6.14. Where to next? 下一步工作

本章，我们大概浏览了下 task ,但这不是 task 的全部，可以详见Chapter 15. More about Tasks 更多关于任务

另外，继续教程 Chapter 7. Java Quickstart 快速开始 Java 和 Chapter 8. Dependency Management Basics 依赖管理的基础知识.md

Chapter 7. Java Quickstart 快速开始 Java

7.1. The Java plugin 关于 Java 插件

Gradle 是一个通用的构建工具，它能构建任何基于你的构建脚本的东西。开箱即用，当然除非你添加代码到你的构建脚本里，不然它不会构建任何东西。

很多 Java 项目都有类似的基本流程：编译 Java 源文件，运行单元测试，创建 JAR 文件。如果你不是把代码从头写到尾，那还能接受。现在有了 Gradle 就不用忍受这些。解决问题的方法就是 插件。插件是 Gradle 配置的扩展，通常是添加配置前的 task。Gradle 装载很多插件，这样可以方便共享。其中，Java 插件 就是添加 task 到 project ，会编译、单元测试你的 Java 代码，并构建进一个 JAR 文件。

Java 插件 是基于约定的。这意味着，该插件定义了 项目 许多方面的的默认值，如 Java 源文件所在的位置。如果你跟随你的项目的约定，你一般不需要在你的构建脚本做太多。Gradle 允许您自定义您的项目，如果你不想或不遵循某种公约。事实上，因为 Java 项目的支持作为一个插件来实现的，你不需要使用所有的插件来构建一个Java项目，如果你不想。

后续章节，我们有许多案例关于 Java 插件、依赖管理、多 project。在这一章中，我们想给你一个初始的想法关于如何使用 Java 插件来构建一个 Java 项目。

7.2. A basic Java project 基本的 Java 项目

为了使用 Java 插件，添加下面代码到构建文件：

Example 7.1. Using the Java plugin

build.gradle

```
apply plugin: 'java'
```

注意，完整的项目源码见*https://github.com/waylau/Gradle-2-User-Guide-Demos* 中 *java/quickstart*

这个就是 定义一个 Java 项目的全部。它会将 Java 插件应用到项目中，并且添加很多 task。

Gradle 会在 src/main/java 目录下寻找产品代码，在 src/test/java 寻找测试代码 。 另外在 src/main/resources 包含了资源的 JAR 文件, src/test/resources 包含了运行测试。所有的输出都在 build 目录下，JAR 在 build/libs 目录下

7.2.1. Building the project 构建项目

在 Java 插件增添了相当多的 task 在 project 中。然而，只有少数的task 是需要在 构建 project 时需要的。最常用的任务是 build task，这就能构建一个完整的 project 。当你运行 gradle build，Gradle 将编译和测试您的代码，并创建一个包含您的主要类和资源的 JAR 文件。

Example 7.2. Building a Java project

执行 gradle build 输出

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

其他有用的 task 有:

clean

删除 build 目录，移除所有构建的文件

assemble

编译打包代码,但不运行单元测试。其他插件带给这个 task 更多特性，比如如果你使用 War 插件，task 将给 project 构建 WAR 文件

check

编译测试你的代码。其他插件带给这个 task 提供更多检查类型。比如，你使用 checkstyle 插件, 这个 task 建辉在你的代码中 执行 Checkstyle

7.2.2. External dependencies 外部依赖

Java 项目经常会有一些外部 JAR 的依赖。为了引用这些 JAR 文件，需要在 Gradle 里面配置。在 Gradle，类似与 JAR 文件将会放在 repository 中。一个 repository 可以被依赖的项目获取到，或者提交项目的拷贝到 repository 中，或者两者都可。比如，我们使用 Maven repository :

Example 7.3. Adding Maven repository

build.gradle

```
repositories {
    mavenCentral()
}
```

我们添加一些依赖，声明了 编译时 需要的依赖和测试时需要的依赖

Example 7.4. Adding dependencies

build.gradle

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

详见Chapter 8. Dependency Management Basics 依赖管理的基础知识

7.2.3. Customizing the project 自定义 项目

在 Java 插件添加属性到您的项目。这些属性通常足在启动时使用默认值。如果他们不适合你，你很容易改他们。让我们看一看我们的示例。在这里，我们将说明我们的 Java 项目的版本号，包括 Java 的版本号。我们也添加一些属性的 JAR 文件清单。

Example 7.5. Customization of MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}
```

Java 插件添加的 task 和 平常的 task 完全一样，在构建文件中声明。这意味着你可以使用任何在前面的章节中自定义这些 task 的机制。例如，您可以设置 task 的性能，添加行为的一个任务，更改 task 的依赖，或替换完全的 task 。在我们的示例，我们将配置测试 task ，这是类型 Test ，增加一个系统属性，当执行测试时：

Example 7.6. Adding a test system property

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

有哪些属性存在？

执行 *gradle properties* 可以列出 *project* 的属性,你可以看到 *Java* 插件添加的属性和他们的默认值

7.2.4. Publishing the JAR file 发布 JAR 文件

需要告诉 Gradle 要发布 JAR 的位置。在 Gradle 中， 产物 比如 JAR 文件等是发布到库中的。我们的例子中是发布到了本地路径。你也可以发布到 远程位置或者多个位置。

Example 7.7. Publishing the JAR file

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

执行 gradle uploadArchives 来发布

7.2.5. Creating an Eclipse project 创建一个 Eclipse project

创建 Eclipse 特点的描述文件，比如 .project，需要添加插件

Example 7.8. Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

执行 gradle eclipse 来生产 Eclipse project 文件。更多 eclipse task 相关内容详见 Chapter 38. The Eclipse Plugin 关于 Eclipse 插件

7.2.6. Summary 总结

下面是完整的示例 build 文件

Example 7.9. Java example - complete build file

build.gradle

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

7.3. Multi-project Java build 多 project 的 Java 构建

下面是一个 多 project 构建的 项目结构：

Example 7.10. Multi-project build - hierarchical layout

```
multiproject/
  api/
  services/webservice/
  shared/
  services/shared/
```

注意，完整的项目源码见*https://github.com/waylau/Gradle-2-User-Guide-Demos* 中 *java/multiproject*

里面包含 4 个 project。 `api` 是产生出 JAR 文件给客户端加载提供给 Java 客户端需要的 XML webservice。 `webservice` 是一个 web 应用返回 XML 。 `shared` 包含了 `api` 、 `webservice` 使用的代码。项目 `services/shared` 包含了 依赖 `shared` 的代码。

7.3.1. Defining a multi-project build 定义 build 文件

配置文件的名字叫 settings.gradle，如下

Example 7.11. Multi-project build - settings.gradle file

settings.gradle

```
include "shared", "api", "services:webservice", "services:shared"
```

详见 Chapter 57. Multi-project Builds 多项目构建

7.3.2. Common configuration 常见配置

有很多常见的配置。我们的示例中使用了 configuration injection （配置注入）。在这里，根项目就像一个容器，subprojects 方法遍历容器中的元素（实例中的 project ），并将指定的配置。这样我们可以很容易地定义所有档案的 manifest 的内容，和一些常见的依赖关系：

Example 7.12. Multi-project build - common configuration

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.11'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

注意，示例中 在 所有 子 project 中应用了 Java 插件。意思是 task 和配置属性将会出现在虽偶有 子 project 中。所以，你可以 在根 project 目录中，运行 gradle build 来编译、测试、将所有 project 打包成 JAR 。

注意，插件只应用在 subprojects 包含的区域，其他根级别的将不适用。

7.3.3. Dependencies between projects 项目间的依赖

在相同的构建里，您可以添加项目之间的依存关系，这样，例如，一个项目的 JAR 文件可以用来编译另外一个项目。在 `api` 构建文件中我们将添加对 `shared` 项目的依赖。由于这种依赖，Gradle 将确保 `shared` 在 `api` 之前获得构建。

Example 7.13. Multi-project build - dependencies between projects

api/build.gradle

```
dependencies {
    compile project(':shared')
}
```

详见 Chapter 57. Multi-project Builds 多项目构建 中 Section 57.7.1, “Disabling the build of dependency projects” 如何禁用这个功能

7.3.4. Creating a distribution 创建发布包

添加发布包，提供给客户端装载

Example 7.14. Multi-project build - distribution file

api/build.gradle

```
task dist(type: Zip) {
    dependsOn spiJar
    from 'src/dist'
    into('libs') {
        from spiJar.archivePath
        from configurations.runtime
    }
}

artifacts {
   archives dist
}
```

7.4. Where to next? 下步工作

你可以查看更多关于 Java 插件 ，见Chapter 23. The Java Plugin 关于 Java 插件。也可以在 https://github.com/waylau/Gradle-2-User-Guide-Demos 中 java 目录下，看到更多 Java 的示例

继续 Chapter 8. Dependency Management Basics 依赖管理的基础知识

Chapter 8. Dependency Management Basics 依赖管理的基础

本章介绍一些 Gradle 依赖管理的基础

8.1. What is dependency management? 什么是依赖管理

大致上，依赖管理是由2块组成。首先，Gradle 需要知道项目构建或者运行的需要是东西。我们把引进的文件称之为 项目的依赖。其次，Gradle 需要构建和上传项目的产物。我们把向外输出的文件称之为项目的发布。现在看下细节：

很多项目不能完全自我包含。他们需要其他项目的产物。比如， 使用 Hibernate ，JDBC driver 或者 Ehcache jars，需要将他们放在我们项目的 classpath， 来实现需要的功能。

这些引进的项目依赖的文件， Gradle 允许你告诉你的项目所需要的依赖，这样项目才能找到他们，在构建的时候使用他们。这些依赖可能要从远程的 Maven 或者 Ivy 下载，放在你的本地的目录，或者需要被其他项目构建（在相同的 多 project 构建中）。我们称之为 dependency resolution （依赖性解析）。

请注意，此功能提供了 Ant 的一个主要优势。与 Ant 相比，你只需要指定需要加载的绝对或相对路径的特定的 jars。在 Gradle，你只是声明依赖的的“名字”， 和其他布局的确定的位置。你可以通过增加 Apache Ivy到 Ant 得到类似的行为，但 Gradle 做得更好。

通常，一个项目的依赖会包含自己的依赖。例如，Hibernate 的核心需要几个其他包在类路径中存在才能运行。所以，当 Gradle 运行你的项目的测试，它也需要找到这些依赖关系，使他们存在。我们称这些 transitive dependencies （过渡依赖）。

大多数项目的主要目的是构建一些文件是在项目中使用。例如，如果你的项目生成 Java 库，你需要建立一个 jar，也许一个源 jar 和一些文档，并将其发布的某个地方。

这些输出文件以发布包的形式。Gradle 还负责这个重要的工作给你。你声明你的项目的发布，Gradle 照顾构建和发布他们。究竟发布什么取决于你想做什么。你可能想将文件复制到本地目录，或将它们上传到一个远程 Maven 或 Ivy 库。或者你可能使用在相同的多 project 的另一个项目文件的构建。我们称这个过程为 publication（发布） 。

8.2. Declaring your dependencies 声明依赖

下面是基本的脚本

Example 8.1. Declaring dependencies

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

这个脚本说明了几个事。首先项目需要 Hibernate core 3.6.7.Final 版本来编译。其中隐含的意思是，Hibernate core 和 他的依赖在运行时是需要的。其次，需要 junit >= 4.0 版本在测试时需要编译。同时 告诉 Gradle 依赖在 Maven central 库 中找。下面详述

8.3. Dependency configurations 项目配置

一个配置是一个简单的命名依赖的集合。我们称它为依赖配置。你可以用它们来声明项目的外部依赖。正如我们将看到的，他们还用声明项目的 发布。

Java 配置定义了一些标准的配置，这些配置在Java 插件使用的 classpath 中，下面是一些列表。详见Chapter 23. The Java Plugin 关于 Java 插件 中 Table 23.5, “Java plugin - dependency configurations“

compile

编译项目的生产源所需的依赖。

runtime

生产类在运行时所需的依赖。默认情况下，还包括编译时的依赖。

testCompile

编译项目的测试源所需的依赖。默认情况下，还包括产品编译类和编译时的依赖。

testRuntime

运行测试所需的依赖。默认情况下，还包括 编译，运行时和测试编译的依赖。

各种插件添加进一步的标准配置。您也可以定义自己的自定义配置，使用你的建构建。请参见Chapter 51. Dependency Manageme Section 51.3, “Dependency configurations” 关于更多自定义依赖配置。

Chapter 9. Groovy Quickstart 快速开始 Groovy

使用 Groovy 插件来构建 Groovy 项目。这个插件继承自 Java 插件，使你的应用具备了编译能力。你的项目可以包含 Groovy 源码，Java 源码，或者两者都包含。在其他各方面，Groovy 项目与我们在Chapter 07. Java Quickstart 快速开始 Java 中所看到的 Java 项目几乎相同 。

9.1. A basic Groovy project一个基本的Groovy项目

让我们来看一个例子。要使用 Groovy 插件，你需要在构建脚本文件当中添加以下内容

Example 9.1. Groovy plugin

build.gradle

```
apply plugin: 'groovy'
```

注意，完整的项目源码见*https://github.com/waylau/Gradle-2-User-Guide-Demos* 中 *groovy/quickstart*

同时会将 Java 插件应用到项目中，如果还没有应用的话。Groovy 插件 继承自 compile task 在 src/main/groovy 目录中查找源文件；且继承了 compileTest task，在 src/test/groovy 目录中查找测试的源文件。这些编译 task 对这些目录使用了联合编译，这意味着它们可以同时包含 Java 和 Groovy 源文件。

Example 9.2. Dependency on Groovy

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
}
```

下面是完整的构建文件：

Example 9.3. Groovy example - complete build file

build.gradle

```
apply plugin: 'eclipse'
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
    testCompile 'junit:junit:4.11'
}
```

运行 `gradle build` 将会对你的项目进行编译，测试和打成 JAR 包。

9.2. Summary 总结

这一章描述了一个很简单的 Groovy 项目。通常情况下，一个真实的项目所需要的不止于此。因为一个 Groovy 项目也是一个 Java 项目，因此你能用 Java 做的事情 Groovy 也能做。

你可以参阅 Chapter 24. The Groovy Plugin 去了解更多关于 Groovy 插件的内容，或在 https://github.com/waylau/Gradle-2-User-Guide-Demos 中 groovy 目录中找到更多的 Groovy 项目示例。

Chapter 10. Web Application Quickstart 快速开始 Web 应用

本章未完，还在进行中

本章介绍了 Gradle 对 Web 应用的相关支持。 Gradle 为 Web 开发提供了两个主要插件,War 插件 和 Jetty 插件。 其中 War 插件继承自 Java 插件,可以用来生成 WAR 文件。Jetty 插件继承自 War 插件 作为工程部署的容器。

10.1. Building a WAR file 构建 WAR 文件

应用 War 插件 来构建 WAR 文件：

Example 10.1. War plugin

build.gradle

```
apply plugin: 'war'
```

注意，完整的项目源码见*https://github.com/waylau/Gradle-2-User-Guide-Demos* 中 *webApplication/quickstart*

同时应用 Java 插件,当你执行 `gradle build` 时,将会编译、测试、打包工程成为一个 WAR 文件。 Gradle 会在 WAR 中 src/main/webapp 下寻找 源文件。编译后的classes文件以及运行时依赖也都会被包含在 WAR 包中，分别在 WEB-INF/classes 和 WEB-INF/lib 目录下。

10.2. Running your web application 运行应用

需要应用 Jetty 插件来运行应用。

Example 10.2. Running web application with Jetty plugin

build.gradle

```
apply plugin: 'jetty'
```

同样需要应用 WAR 插件，当你执行 `gradle jettyRun` 时,将会运行应用在一个内嵌的 Jetty Web 容器里。运行 `gradle jettyRunWar` 将会构建成 WAR 文件，接着运行在内嵌 的 Web 容器。

TODO:url,端口,以及源文件位置都可以在脚本中进行指定修改并重载。

Groovy web 应用

在一个项目中你可以采用多个插件。比如你可以在 *web* 项目中同时使用*War* 插件和 *Groovy* 插件来构建基于 *web* 应用的 *Groovy*。适当的 *Groovy* 库将被添加到 *WAR* 的文件中。

10.3. Summary 总结

了解更多关于 War 插件 和 Jetty 插件的请参阅Chapter 26. The War Plugin 关于 War 插件以及 Chapter 28. The Jetty Plugin 关于 Jetty 插件。你可以在https://github.com/waylau/Gradle-2-User-Guide-Demos 中 webApplication 下找到更多示例.

Chapter 11. Using the Gradle Command-Line 使用 Gradle 命令行

本章介绍了 Gradle 命令行的基本知识。正如在前面的章节里你所见到的， 调用 gradle 命令来执行构建。

11.1. Executing multiple tasks 执行多 task

同个构建可以执行多个 task ,通过再命令行 列出每个 task。举例，命令 `gradle compile test` 将会执行 compile 和 test 两个 task。 Gradle 将会按顺序执行 命令行每个列出的 task，并且执行每个 task 的依赖。每个任务仅执行一次，不管它是如何被包含在构建中：无论是在命令行中指定，或作为另一个 task 的依赖，或两者都是。让我们看一个例子。

下面定义了4个 task。 dist 和 test 都依赖于 compile 。执行 `gradle dist test` ，compile 将会仅仅被执行一次。

Figure 11.1. Task dependencies

Example 11.1. Executing multiple tasks

build.gradle

```
task compile << {
    println 'compiling source'
}

task compileTest(dependsOn: compile) << {
    println 'compiling unit tests'
}

task test(dependsOn: [compile, compileTest]) << {
    println 'running unit tests'
}

task dist(dependsOn: [compile, test]) << {
    println 'building the distribution'
}
```

执行 gradle dist test 输出

```
> gradle dist test
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

每个 task 只执行一次，所以 `gradle test test` 跟 `gradle test` 执行结果一样。

11.2. Excluding tasks 排除 task

可以通过 -x 命令行来排除 task 被执行。如下：

Example 11.2. Excluding tasks

执行 gradle dist -x test 输出

```
> gradle dist -x test
:compile
compiling source
:dist
building the distribution

BUILD SUCCESSFUL
```

Total time: 1 secs

可以看到 ，test 并未执行，即使它是 dist 的依赖。同时注意到， test 的依赖，如 compileTest 也未执行。这些 test 的依赖如果是被其他 task 所需要的话，如 compile 仍会执行。

11.3. Continuing the build when a failure occurs 发生故障时继续构建

默认情况下，只要任何 task 失败，Gradle 将中止执行。这使得构建更快地完成，但隐藏了其他可能发生的故障。为了发现在一个单一的构建中多个可能发生故障的地方，你可以使用 --continue 选项。

通过执行 --continue ，Gralde 会执行每一个 task ，当那个 task 所有的依赖都无故障的执行完成, 而不是一旦出现错误就会中断执行。所有故障信息都会在最后报告出来。

一旦某个 task 执行失败,那么所有依赖于该 task 的后面的 task 都不会被执行，因为这样做不安全。例如，在测试时，当编译代码失败则测试不会执行。因为 测试 task 将取决于 编译 task（不管是直接或间接）。

11.4. Task name abbreviation 任务名称缩写

当你试图执行某个 task 的时候,无需输入 task 的全名.只需提供足够的可以唯一区分出该 task 的字符即可。例如,上面的例子你也可以这么写, 用 `gradle di` 来直接调用 dist 。

Example 11.3. Abbreviated task name

执行 gradle di

```
> gradle di
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

同时也可以应用在驼峰的 task 名称。如，执行 compileTest 时，运行 gradle compTest 或者 gradle cT 都可以。

Example 11.4. Abbreviated camel case task name

执行 gradle cT

```
> gradle cT
:compile
compiling source
:compileTest
compiling unit tests

BUILD SUCCESSFUL

Total time: 1 secs
```

同时，也可以应用在 -x 命令选项。

11.5. Selecting which build to execute 选择要执行的构建

调用 gradle 命令时,默认情况下总是会在当前目录下寻找构建文件（译者注：首先会寻找当前目录下的 build.gradle 文件,以及根据settings.gradle 中的配置寻找子项目的 build.gradle ）。可以使用 -b 参数选择其他的构建文件,并且当你使用此参数时 settings.gradle 将不会被使用,看下面的例子:

Example 11.5. Selecting the project using a build file

subdir/myproject.gradle

```
task hello << {
    println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

执行 gradle -q -b subdir/myproject.gradle hello 输出

```
> gradle -q -b subdir/myproject.gradle hello
using build file 'myproject.gradle' in 'subdir'.
```

或者，您可以使用 -p 选项来指定要使用的项目目录。多 project 的构建时应使用 -p 选项来代替 -b 选项。

Example 11.6. Selecting the project using project directory

执行 gradle -q -p subdir hello 输出

```
> gradle -q -p subdir hello
using build file 'build.gradle' in 'subdir'.
```

11.6. Obtaining information about your build 获取构建信息

Gradle 提供了许多内置 task 来收集构建信息。这些内置 task 对于了解依赖结构以及解决问题都是很有帮助的.

了解更多,可以参阅Chapter 41. The Project Report Plugin 关于 Project Report 插件,可以为你的项目添加构建报告

11.6.1. Listing projects 项目列表

执行 `gradle projects` 会为你列出选中项目的子项目列表.如下例.

Example 11.7. Obtaining information about projects

执行 gradle -q projects 输出

```
> gradle -q projects
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

这份报告展示了每个项目的描述信息。当然你可以在项目中用 description属性来指定这些描述信息。

Example 11.8. Providing a description for a project

build.gradle

```
description = 'The shared API for the application'
```

11.6.2. Listing tasks 任务列表

执行 `gradle tasks` 会列出项目中所有 task。这份报告显示项目中所有的默认 task 以及每个 task 的描述。如下

Example 11.9. Obtaining information about tasks

执行 gradle -q tasks 输出

```
> gradle -q tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'projectReports'. [incub
ating]
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project 'p
rojectReports'.
help - Displays a help message.
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the di
splayed tasks may belong to subprojects).

To see all tasks and more detail, run with --all.
```

默认情况下,这只会显示那些被分组的 task.你可以通过为 task 设置group 属性和 description 来把这些信息展示到报告中

Example 11.10. Changing the content of the task report

build.gradle

```
dists {
    description = 'Builds the distribution'
    group = 'build'
}
```

当然你也可以用 --all 参数来收集更多 task 信息。这报告列出项目中所有被主 task 的分组的 task 以及 task 之间的依赖关系。下面是示例

Example 11.11. Obtaining more information about tasks

执行 gradle -q tasks --all 输出

```
> gradle -q tasks --all
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'projectReports'. [incub
ating]
api:components - Displays the components produced by project ':api'. [incubating]
webapp:components - Displays the components produced by project ':webapp'. [incubating
]
dependencies - Displays all dependencies declared in root project 'projectReports'.
api:dependencies - Displays all dependencies declared in project ':api'.
webapp:dependencies - Displays all dependencies declared in project ':webapp'.
dependencyInsight - Displays the insight into a specific dependency in root project 'p
rojectReports'.
api:dependencyInsight - Displays the insight into a specific dependency in project ':a
pi'.
webapp:dependencyInsight - Displays the insight into a specific dependency in project
':webapp'.
help - Displays a help message.
api:help - Displays a help message.
webapp:help - Displays a help message.
projects - Displays the sub-projects of root project 'projectReports'.
api:projects - Displays the sub-projects of project ':api'.
webapp:projects - Displays the sub-projects of project ':webapp'.
properties - Displays the properties of root project 'projectReports'.
api:properties - Displays the properties of project ':api'.
```

```
webapp:properties - Displays the properties of project ':webapp'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the di
splayed tasks may belong to subprojects).
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

11.6.3. Show task usage details 显示 task 使用细节

执行 gradle help --task someTask 可以获取到 task 的详细信息，或者多项目构建中相同 task 名称的所有 task 的信息,如下

Example 11.12. Obtaining detailed help for tasks

执行 gradle -q help --task libs 输出

```
> gradle -q help --task libs
Detailed task information for libs

Paths
     :api:libs
     :webapp:libs

Type
     Task (org.gradle.api.Task)

Description
     Builds the JAR
```

这些结果包含了完整的 task 的路径、类型、可能的命令行选项以及描述信息等.

11.6.4. Listing project dependencies 依赖列表

执行 gradle dependencies 会列出项目的依赖列表,所有依赖会根据任务区分,以树型结构展示出来。如下

Example 11.13. Obtaining information about dependencies

执行 gradle -q dependencies api:dependencies webapp:dependencies 输出

```
> gradle -q dependencies api:dependencies webapp:dependencies
------------------------------------------------------------
Root project
------------------------------------------------------------

No configurations

------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

compile
\--- org.codehaus.groovy:groovy-all:2.3.6

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3

------------------------------------------------------------
Project :webapp - The Web application implementation
------------------------------------------------------------

compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.3.6
\--- commons-io:commons-io:1.2

testCompile
No dependencies
```

由于依赖的报告可以变得较大，可以使用特定的配置来限制到一个有用的报告。可以通过 --configuration 可选参数来实现。

Example 11.14. Filtering dependency report by configuration

执行 gradle -q api:dependencies --configuration testCompile 输出为

```
> gradle -q api:dependencies --configuration testCompile
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3
```

11.6.5. Getting the insight into a particular dependency 查看特定依赖

执行 gradle dependencyInsight 可以查看指定的依赖情况，如下

Example 11.15. Getting the insight into a particular dependency

执行 gradle -q webapp:dependencyInsight --dependency groovy --configuration compile 输出

```
> gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
org.codehaus.groovy:groovy-all:2.3.6
\--- project :api
     \--- compile
```

这对于分辨依赖、了解依赖关系、了解为何选择此版本作为依赖十分有用。了解更多请参阅 DependencyInsightReportTask 类的 API

内建的 dependencyInsight 是'Help' task 分组中的一个。这项 task 需要进行依赖和配置文件的配置才可以。该报告寻找那些与定依赖规范指定的配置匹配的的依赖。如果应用了 Java 相关的插件，该dependencyinsight task 是预先经过 'compile' 配置，因为它通常依赖我们感兴趣的编译。你应该指定您感兴趣的依赖,通过命令行 '--dependency'选项。如果你不喜欢默认的，你可以选择通过 '--configuration' 选项来配置。更多信息见DependencyInsightReportTask 类的API文档。

11.6.6. Listing project properties 项目属性列表

执行 gradle properties 可以获取项目所有属性列表，如下

Example 11.16. Information about properties

执行 gradle -q api:properties 输出

```
> gradle -q api:properties
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@1234
5
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@12345
baseClassLoaderScope: org.gradle.api.internal.initialization.DefaultClassLoaderScope@1
2345
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.gradl
e
```

11.6.7. Profiling a build

--profile 命令选项可以记录一些构建期间的信息并保存到 build/reports/profile 目录下并且以构建时间命名这些文件

该报告列出总时间和在配置和 task 的执行 阶段的细节。并以时间大小倒序排列，并且记录了任务的执行情况

如果采用了 buildSrc 构建,那么在 buildSrc/build 下同时也会给 buildSrc 生成一份日志记录

11.7. Dry Run 干跑

有时可能你只想知道某个 task 在一个 task 集中按顺序执行的结果,但并不想实际执行这些 task 。那么你可以用 -m 选项。例如 执行 `gradle -m clean compile` 将会看到所有的作为 clean 和 compile 一部分的 task 会被执行。这与 task 可以形成互补,让你知道哪些 task 可以用于执行。

11.8. Summary 总结

本章看到了命令行的一部分。更多详见 Appendix D. Gradle Command Line 命令行

Chapter 12. Using the Gradle Graphical User Interface 使用 Gradle 图形化用户界面

除了支持传统的命令行界面，Gradle 也提供了一个图形化用户界面（GUI）。这是一个独立的用户界面，可以通过加上 --gui 选项来启动。

Example 12.1. Launching the GUI

```
gradle --gui
```

注意：此命令行窗口被将锁定，直到 Gradle GUI 被关闭。如果是在 *nix 系统下，则可以通过 (gradle --gui&) 让它作为后台任务运行。

如果你在你的 Gradle 项目目录下运行 Gradle GUI，你应该会看到一个task 树。

Figure 12.1. GUI Task Tree

最好是从 Gradle 项目目录运行此命令，这样对 UI 的设置就可以存储在你的项目目录中。当然，你也可以先运行它，然后通过在 UI 中的设置Setup 选项卡，改变工作目录。

这个 UI 上面是4个选项卡，下面则是输出窗口。

12.1. Task Tree 任务树

Task 树显示了所有项目和它们的 task 的层次结构。双击一个 task 可以执行它。

这里还提供了一个过滤器，可以把比较少用的 task 隐藏。你可以通过过滤器（Filter）按钮切换是否进行过滤。通过编辑过滤器，你可以对哪些任务和项目要显示进行配置。隐藏的任务显示为红色。注意：新创建的任务默认情况下是显示状态（而不是隐藏状态）

任务树的上下文菜单会提供以下选项：

- 执行忽略依赖关系。这使得重新构建时不去依赖项目（与 -a 选项一样）
- 将任务添加到收藏夹（见收藏夹 Favourites 选项卡）
- 隐藏选择的任务。这将会把它们添加到过滤器中。
- 编辑 build.gradle 文件。注意：该操作需要 Java 1.6 或更高的版本，并且要求在你的操作系统中关联 gradle 文件。

12.2. Favorites 收藏夹

收藏夹选项卡用来储存经常执行的命令。这些命令可以是复杂的命令（只要它们符合 Gradle 的语法），你可以给它们设置一个显示名称。它用于创建一个自定义的命令，来显示地跳过测试，文档，例子。你可以称之为“快速构建”。

你可以根据自己的喜好，对收藏夹进行排序，甚至可以把它们导出到磁盘，并在其他地方导入。如果你在编辑它们的时候，选上“始终显示实时输出”，它只有在你选上“当发生错误时才显示输出”时有效。它会始终强制显示输出。

12.3. Command Line 命令行

命令行选项卡是直接执行单个的 Gradle 命令的地方。你只需要输入在 'gradle' 后面经常输入的命令即可 。它也对要添加到收藏夹的命令提供了先去尝试的地方。

12.4. Setup 设置

用来设置常用的配置

Figure 12.2. GUI Setup

Current Directory (当前目录)

定义了你的 Gradle 项目（通常是 build.gradle 所在的位置）的根目录。

Stack Trace Output (堆栈跟踪输出)

这决定了当出现错误时，有多少信息定到堆栈跟踪。注意：如果你在命令行或收藏夹选项卡上指定了堆栈跟踪级别，将会覆盖这里的设置。

Only Show Output When Errors Occur(只在出现错误时显示输出)

启用此选项将在 task 执行时隐藏任何输出，除非构建失败。

Use Custom Gradle Executor - Advanced feature(使用自定义的 Gradle 执行器 - 高级功能)

这为你提供了启动 Gradle 命令行的替代方法。这是很有用的。如果你的项目需要在另一个批处理文件或 shell 脚本中做一些额外的配置（比如指定一个初始化脚本）。

Chapter 13. Writing Build Scripts 编写构建脚本

本章着眼于一些编写构建脚本的详细信息。

13.1. The Gradle build language 构建语言

Gradle 提供一种 domain specific language （领域特定语言）或者说是 DSL，来描述构建。这种构建语言基于 Groovy 中，并进行了一些补充，使其易于描述构建。

构建脚本可以包含任何 Groovy 语言的元素（除了声明标签任何语言元素）。Gradle 假定每个构建脚本使用 UTF-8 编码。

13.2. The Project API 项目 API

在Chapter 07. Java Quickstart 快速开始 Java的教程中，我们使用了 apply() 方法。这方法从何而来？我们之前说在 Gradle 中构建脚本定义了一个 project 。在构建的每一个 project，Gradle 创建了一个 Project 类型的实例，并在构建脚本中关联此 Project 对象。当构建脚本执行时，它会配置此Project 对象：

- 在构建脚本中，你所调用的任何一个方法，如果在构建脚本中未定义，它将被委托给 Project 对象。
- 在构建脚本中，你所访问的任何一个属性，如果在构建脚本里未定义，它也会被委托给 Project 对象。

下面我们来试试这个，试试访问 Project 对象的 name 属性。

Example 13.1. Accessing property of the Project object

build.gradle

```
println name
println project.name
```

执行 gradle -q check

```
> gradle -q check
projectApi
projectApi
```

这两个 println 语句打印出相同的属性。在生成脚本中未定义的属性，第一次使用时自动委托到P roject 对象。其他语句使用了在任何构建脚本中可以访问的 project 属性，则返回关联的 Project 对象。只有当您定义的属性或方法 Project 对象的一个成员相同名字时，你才需要使用project 属性。

获取有关编写构建脚本帮助

不要忘记您的构建脚本是简单的 *Groovy* 代码，并驱动着 *Gradle API*。并且 *Project* 接口是您在 *Gradle API* 中访问一切 的入点。所以，如果你想知道什么 '标签（*tag*）' 在构建脚本中可用，您可以去看项目接口的文档。

13.2.1. Standard project properties 标准 project 属性

Project 对象提供了一些在构建脚本中可用的标准的属性。下表列出了常用的几个属性。

名称	类型	默认值
`project`	`Project`	The `Project实例`
`name`	`String`	项目目录的名称
`path`	`String`	项目的绝对路径
`description`	`String`	项目的描述
`projectDir`	`File`	包含生成脚本的目录
`buildDir`	`File`	`projectDir/build`
`group`	`Object`	`未指定`
`version`	`Object`	`未指定`
`ant`	`AntBuilder`	An `AntBuilder实例`

13.3. The Script API 脚本 API

当 Gradle 执行一个脚本时，它将脚本编译为一个实现了 Script 接口的类。这意味着所有由该 Script 接口声明的属性和方法在您的脚本中是可用的。

13.4. Declaring variables 声明变量

有两类可以在生成脚本中声明的变量： 局部变量和额外属性。

13.4.1. Local variables 局部变量

局部变量是用 def 关键字声明的。它们只在定义它们的范围内可以被访问。局部变量是 Groovy 语言底层的一个特征。

Example 13.2. Using local variables

build.gradle

```
def dest = "dest"

task copy(type: Copy) {
    from "source"
    into dest
}
```

13.4.2. Extra properties 额外属性

Gradle 的域模型中，所有增强的对象都可以容纳用户定义的额外的属性。这包括但并不限于 project、task 和源码集。额外的属性可以通过所属对象的 ext 属性进行添加，读取和设置。或者，可以使用 ext块同时添加多个属性。

Example 13.3. Using extra properties

build.gradle

```
apply plugin: "java"

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}

task printProperties << {
    println springVersion
    println emailNotification
    sourceSets.matching { it.purpose == "production" }.each { println it.name }
}
```

执行 gradle -q printProperties

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

在此示例中， 一个 ext 代码块将两个额外属性添加到 project 对象中。此外，通过将 ext.purpose设置为 null（null是一个允许的值），一个名为 purpose 的属性被添加到每个源码集。一旦属性被添加，他们就可以像预定的属性一样被读取和设置。

通过添加属性所要求特殊的语法，Gradle 可以在你试图设置 （预定义的或额外的） 的属性，但该属性拼写错误或不存在时马上失败。额外属性在任何能够访问它们所属的对象的地方都可以被访问，这使它们有着比局部变量更广泛的作用域。父项目上的额外属性，在子项目中也可以访问。

有关额外属性和它们的 API 的详细信息，请参阅ExtraPropertiesExtension 类的 API。

13.5. Some Groovy basics 一些 Groovy 的基础

Groovy 提供了用于创建 DSL 的大量特点，并且 Gradle 构建语言利用了这些特点。了解构建语言是如何工作的，将有助于你编写构建脚本，特别是当你开始写自定义插件和 task 的时候。

13.5.1. Groovy JDK

Groovy 对 Java 的标准类增加了很多有用的方法。例如， Iterable 新增的each方法，会对 Iterable 的元素进行遍历：

Example 13.4. Groovy JDK methods

build.gradle

```
// Iterable gets an each() method
configurations.runtime.each { File f -> println f }
```

在 http://groovy.codehaus.org/groovy-jdk/ 查看详情

13.5.2. Property accessors属性访问器

Groovy 会自动地把一个属性的引用转换为对适当的 getter 或 setter 方法的调用。

Example 13.5. Property accessors

build.gradle

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

13.5.3. Optional parentheses on method calls 括号可选的方法调用

调用方法时括号是可选的。

Example 13.6. Method call without parentheses

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

13.5.4. List and map literals

Groovy 提供了一些定义 List 和 Map 实例的快捷写法。两种类型都是简单的 literal，但 map literal 有一些有趣的曲折。

例如，“apply” 方法（如你通常应用插件）需要 map 参数。然而，当你有一个像 “apply plugin:'java'”，你实际上并没有使用 map literal ，你实际上使用“命名的参数”，这几乎是和 map literal 相同的语法（不包包装器）。命名参数列表将被转换成一个 map 当方法被调用时，但它没有在开始作为一个 map。

Example 13.7. List and map literals

build.gradle

```
// List literal
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

// Map literal.
Map<String, String> map = [key1:'value1', key2: 'value2']

// Groovy will coerce named arguments
// into a single map argument
apply plugin: 'java'
```

13.5.5. Closures as the last parameter in a method 作为方法最后一个参数的闭包

Gradle DSL 在很多地方使用闭包。你可以在这里查看更多有关闭包的资料。当方法的最后一个参数是一个闭包时，你可以把闭包放在方法调用之后：

Example 13.8. Closure as method parameter

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

13.5.6. Closure delegate 闭包委托

每个闭包都有一个委托对象，Groovy 使用它来查找变量和方法的引用，而不是作为闭包的局部变量或参数。Gradle 在配置闭包中使用到它，把委托对象设置为被配置的对象。

Example 13.9. Closure delegates

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.11')
    delegate.testCompile('junit:junit:4.11')
}
```

Chapter 14. Tutorial - 'This and That' 教程-这个那个

14.1. Directory creation 创建目录

多个 task 依赖于现存的目录，这是常见的情况。当然，你可以在 task 前添加 `mkdir` 但这不是好办法，因为你只需要一次，却要不断重复代码序列（看看 DRY principle ）。好的做法是在 task 间使用 dependsOn 来重用 task 创建目录

Example 14.1. Directory creation with mkdir

build.gradle

```
def classesDir = new File('build/classes')

task resources << {
    classesDir.mkdirs()
    // do something
}
task compile(dependsOn: 'resources') << {
    if (classesDir.isDirectory()) {
        println 'The class directory exists. I can operate'
    }
    // do something
}
```

执行 gradle -q compile 输出

```
> gradle -q compile
The class directory exists. I can operate
```

14.2. Gradle properties and system properties 属性

Gradle 提供了许多方式将属性添加到您的构建中。 从 Gradle 启动的 JVM，你可以使用 -D 命令行选项向它传入一个系统属性。 Gradle 命令的 -D 选项和 java 命令的 -D 选项有着同样的效果。

此外，您也可以通过属性文件直接向您的 project 对象添加属性。您可以把一个 gradle.properties 文件放在 Gradle 的用户主目录（默认如果不是 USER_HOME /.gradle 设置的话，就由 “GRADLE_USER_HOME” 环境变量定义），或您的项目目录中。对于多项目构建，您可以将 gradle.properties 文件放在任何子项目的目录中。通过 project 对象，可以访问到 gradle.properties 里的属性。用户的主目录中的属性文件比项目目录中的属性文件更先被访问到。

你还可以通过使用 -P 命令行选项来直接向您的 project 对象添加属性。

也可以通过 特别命名的系统属性 或者环境属性把属性设置 project 属性。这个特性非常有用，通常出于安全原因，当你在持续集成的服务器没有管理员权限时，对于设置属性值你是不可见的。在这种情况下，你就不能使用 -P 选项，也不能修改系统级别的配置文件。正确的策略是，要改变你的持续集成配置建设工作，增加一个环境变量设置符合预期的模式。这不会对系统正常的用户可见的。（ Jenkins, Teamcity, 或者 Bamboo 是这些 CI （Continuous Integration 持续集成）服务商，提供这些功能）

如果环境变量名称类似与 ORG_GRADLE_PROJECT_prop=somevalue, Gradle 将会设置 prop 属性到你的 project 对象，值就是 somevalue 。Gradle 也提供了系统属性的支持，但有着不同命名方式，类似的 org.gradle.project.prop

也可在 gradle.properties 设置系统属性。如果此类文件中的属性有一个 `systemProp.` 的前缀，像 `systemProp.propName`,该属性和它的值会被添加到系统属性，且不带此前缀。在多 project 构建中，除了在根项目之外的任何项目里的 `systemProp.` 属性集都将被忽略。也就是，只有根 project 的 gradle.properties 文件里的 `systemProp.` 前缀的 属性会被作为系统属性。

Example 14.2. Setting properties with a gradle.properties file

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
    println System.properties['system']
}
```

执行 gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps 输出

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.
systemProjectProp=systemPropertyValue printProps
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

14.2.1. Checking for project properties 检查 project 属性

当你要使用一个变量时，你可以仅通过其名称在构建脚本中访问一个项目的属性。如果此属性不存在，则会引发异常，并且构建失败。如果您的构建脚本依赖于一些可选属性，而这些属性用户可能在比如 gradle.properties 文件中设置，您就需要在访问它们之前先检查它们是否存在。你可以通过使用方法 hasProperty('propertyName') 来进行检查，它返回 true 或 false。

14.3. Configuring the project using an external build script 使用外部构建脚本配置项目

你可以使用一个外部构建脚本配置当前 project 。所有的 Gradle 构建语言都可用在外部脚本。。您甚至可以在外部脚本中应用其他脚本。

Example 14.3. Configuring the project using an external build script

build.gradle

```
apply from: 'other.gradle'
other.gradle

println "configuring $project"
task hello << {
    println 'hello from other script'
}
```

执行 gradle -q hello 输出

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

14.4. Configuring arbitrary objects 配置任意对象

您可以用以下非常易理解的方式配置任意对象。

Example 14.4. Configuring arbitrary objects

build.gradle

```
task configure << {
    def pos = configure(new java.text.FieldPosition(10)) {
        beginIndex = 1
        endIndex = 5
    }
    println pos.beginIndex
    println pos.endIndex
}
```

执行 gradle -q configure 输出

```
> gradle -q configure
1
5
```

14.5. Configuring arbitrary objects using an external script 使用外部脚本配置任意对象

Example 14.5. Configuring arbitrary objects using a script

build.gradle

```
task configure << {
    def pos = new java.text.FieldPosition(10)
    // Apply the script
    apply from: 'other.gradle', to: pos
    println pos.beginIndex
    println pos.endIndex
}
```

other.gradle

执行 gradle -q configure 输出

```
> gradle -q configure
1
5
```

14.6. Caching 缓存

为了提高响应速度，默认情况下 Gradle 会缓存所有已编译的脚本。这包括所有构建脚本，初始化脚本和其他脚本。你第一次运行一个项目构建时， Gradle 会创建 .gradle 目录，用于存放已编译的脚本。下次你运行此构建时， 如果该脚本自它编译后没有被修改，Gradle 会使用这个已编译的脚本。否则该脚本会重新编译，并把最新版本存在缓存中。如果您通过 --recompile-scripts 选项运行 Gradle ，会丢弃缓存的脚本，然后重新编译此脚本并将其存在缓存中。通过这种方式，您可以强制 Gradle 重新生成缓存

Chapter 15. More about Tasks 更多关于任务

在入门教程中（Chapter 06. Build Script Basics 构建脚本的基础识），已经学到了如何创建简单 task。之后您还学习了如何将其他行为添加到这些 task 中，同时你已经学会了如何创建 task 之间的依赖。这都是简单的 task 。但 Gradle 让 task 的概念更深远。Gradle 支持增强的 task，也就是，有自己的属性和方法的 task 。这是真正的与你所使用的 Ant target（目标）的不同之处。这种增强的任务可以由你提供，或由 Gradle 构建。

15.1. Defining tasks 定义任务

在（Chapter 06. Build Script Basics 构建脚本的基础识）中我们已经看到如何通过关键字这种风格来定义 task 。在某些情况中，你可能需要使用这种关键字风格的几种不同的变式。例如，在表达式中不能用这种关键字风格。

Example 15.1. Defining tasks

build.gradle

```
task(hello) << {
    println "hello"
}

task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

也可以使用字符串作为 task 名称

Example 15.2. Defining tasks - using strings for task names

build.gradle

```
task('hello') <<
{
    println "hello"
}

task('copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

你可能更愿意使用另外一种替代的语法来定义任务：

Example 15.3. Defining tasks with alternative syntax

build.gradle

```
tasks.create(name: 'hello') << {
    println "hello"
}

tasks.create(name: 'copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

15.2. Locating tasks 定位任务

你经常需要在构建文件中查找你所定义的 task，例如，为了去配置或是使用它们作为依赖。对这样的情况，有很多种方法。首先，每个 task 都可作为 project 的一个属性，并且使用 task 名称作为这个属性名称：

Example 15.4. Accessing tasks as properties

build.gradle

```
task hello

println hello.name
println project.hello.name
```

task 也可以通过 task 集合来访问

Example 15.5. Accessing tasks via tasks collection

build.gradle

```
task hello

println tasks.hello.name
println tasks['hello'].name
```

您可以从任何 project 中，使用 tasks.getByPath() 方法获取 task 路径并且通过这个路径来访问 task。你可以用 task 名称，相对路径或者是绝对路径作为参数调用 getByPath() 方法。

Example 15.6. Accessing tasks by path

build.gradle

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

执行 gradle -q hello

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA:hello
```

详见 TaskContainer 关于更多定位 task 的选项

15.3. Configuring tasks 配置任务

作为一个例子，让我们看看由 Gradle 提供的 Copy task。若要创建Copy task ，您可以在构建脚本中声明：

Example 15.7. Creating a copy task

build.gradle

```
task myCopy(type: Copy)
```

上面的代码创建了一个什么都没做的复制 task 。可以使用它的 API 来配置这个任务 （见 Copy）。下面的示例演示了几种不同的方式来实现相同的配置。

要明白，意识到这项任务的名称是 “myCopy”，但它的类型是“Copy”。你可以有多个同一类型的 task ，但具有不同的名称。你会发现这给你一个很大的权力来实现横切关注点在一个特定类型的所有 task。

Example 15.8. Configuring a task - various ways

build.gradle

```
Copy myCopy = task(myCopy, type: Copy)
myCopy.from 'resources'
myCopy.into 'target'
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

这类似于我们通常在 Java 中配置对象的方式。您必须在每一次的配置语句重复上下文（myCopy）。这显得很冗余并且很不好读。

还有另一种配置任务的方式。它也保留了上下文，且可以说是可读性最强的。它是我们通常最喜欢的方式。

Example 15.9. Configuring a task - with closure

build.gradle

```
task myCopy(type: Copy)

myCopy {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

这种方式适用于任何任务。该例子的第 3 行只是 tasks.getByName() 方法的简洁写法。特别要注意的是，如果您向 getByName() 方法传入一个闭包，这个闭包的应用是在配置这个任务的时候，而不是任务执行的时候。

您也可以在定义一个任务的时候使用一个配置闭包。

Example 15.10. Defining a task with closure

build.gradle

```
task copy(type: Copy) {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

15.4. Adding dependencies to a task 给任务添加依赖

定义任务的依赖关系有几种方法。在第 6.5 章节，"任务依赖"中，已经向你介绍了使用任务名称来定义依赖。任务的名称可以指向同一个项目中的任务，或者其他项目中的任务。要引用另一个项目中的任务，你需要把它所属的项目的路径作为前缀加到它的名字中。下面是一个示例，添加了从projectA:taskX 到 projectB:taskY 的依赖关系：

Example 15.11. Adding dependency on task from another project

build.gradle

```
project('projectA') {
    task taskX(dependsOn: ':projectB:taskY') << {
        println 'taskX'
    }
}

project('projectB') {
    task taskY << {
        println 'taskY'
    }
}
```

执行 gradle -q taskX

```
> gradle -q taskX
taskY
taskX
```

您可以使用一个 Task 对象而不是任务名称来定义依赖，如下：

Example 15.12. Adding dependency using task object

build.gradle

```
task taskX << {
    println 'taskX'
}

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

执行 gradle -q taskX

```
> gradle -q taskX
taskY
taskX
```

对于更高级的用法，您可以使用闭包来定义 task 依赖。在计算依赖时，闭包会被传入正在计算依赖的任务。这个闭包应该返回一个 Task 对象或是Task 对象的集合，返回值会被作为这个 task 的依赖项。下面的示例是从taskX 加入了 project 中所有名称以 lib 开头的 task 的依赖

Example 15.13. Adding dependency using closure

build.gradle

```
task taskX << {
    println 'taskX'
}

taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

task notALib << {
    println 'notALib'
}
```

执行 gradle -q taskX

```
> gradle -q taskX
lib1
lib2
taskX
```

更多关于 task 依赖 ，见 Task API

15.5. Ordering tasks 排序任务

任务排序还是一个孵化中的功能。请注意此功能在以后的 Gradle 版本中可能会改变。

在某些情况下，控制两个任务的执行的顺序，而不引入这些任务之间的显式依赖，是很有用的。任务排序和任务依赖之间的主要区别是，排序规则不会影响那些任务的执行，而仅将执行的顺序。

任务排序在许多情况下可能很有用：

- 强制任务顺序执行： 如，'build' 永远不会在 'clean' 前面执行。
- 在构建中尽早进行构建验证：如，验证在开始发布的工作前有一个正确的证书。
- 通过在长久验证前运行快速验证以得到更快的反馈：如，单元测试应在集成测试之前运行。
- 一个任务聚合了某一特定类型的所有任务的结果：如，测试报告任务结合了所有执行的测试任务的输出。

有两种排序规则是可用的："必须在之后运行"和"应该在之后运行"。

通过使用 “必须在之后运行”的排序规则，您可以指定 taskB 必须总是运行在 taskA 之后，无论 taskA 和 taskB 这两个任务在什么时候被调度执行。这被表示为 taskB.mustRunAfter(taskA) 。“应该在之后运行”的排序规则与其类似，但没有那么严格，因为它在两种情况下会被忽略。首先是如果使用这一规则引入了一个排序循环。其次，当使用并行执行，并且一个任务的所有依赖项除了任务应该在之后运行之外所有条件已满足，那么这个任务将会运行，不管它的“应该在之后运行”的依赖项是否已经运行了。当倾向于更快的反馈时，会使用“应该在之后运行”的规则，因为这种排序很有帮助但要求不严格。

目前使用这些规则仍有可能出现 taskA 执行而 taskB 没有执行，或者taskB 执行而 taskA 没有执行。

Example 15.14. Adding a 'must run after' task ordering

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.mustRunAfter taskX
```

执行 gradle -q taskY taskX

```
> gradle -q taskY taskX
taskX
taskY
```

Example 15.15. Adding a 'should run after' task ordering

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.shouldRunAfter taskX
```

执行 gradle -q taskY taskX

```
> gradle -q taskY taskX
taskX
taskY
```

在上面的例子中，它仍有可能执行 taskY 而不会导致 taskX 也运行：

Example 15.16. Task ordering does not imply task execution

执行 gradle -q taskY

```
> gradle -q taskY
taskY
```

如果想指定两个任务之间的“必须在之后运行”和“应该在之后运行”排序，可以使用 Task.mustRunAfter()) 和 Task.shouldRunAfter()) 方法。这些方法接受一个任务实例、任务名称或 Task.dependsOn()) 所接受的任何其他输入作为参数。

请注意"B.mustRunAfter(A)"或"B.shouldRunAfter(A)"并不意味着这些任务之间的任何执行上的依赖关系：

它是可以独立地执行任务 A 和 B 的。

- 排序规则仅在这两项任务计划执行时起作用。
- 当--continue参数运行时，可能会是 A 执行失败后 B 执行了。

如之前所述，如果“应该在之后运行”的排序规则引入了排序循环，那么它将会被忽略。

Example 15.17. A 'should run after' task ordering is ignored if it introduces an ordering cycle

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
task taskZ << {
    println 'taskZ'
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

执行 gradle -q taskX

```
> gradle -q taskX
taskZ
taskY
taskX
```

15.6. Adding a description to a task 给任务添加描述

可以给任务添加描述，这个描述将会在 task 执行时显示。

Example 15.18. Adding a description to a task

build.gradle

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

15.7. Replacing tasks 替换任务

有时您想要替换一个任务。例如，您想要把通过 Java 插件添加的一个任务与不同类型的一个自定义任务进行交换。你可以这样实现：

Example 15.19. Overwriting a task

build.gradle

```
task copy(type: Copy)

task copy(overwrite: true) << {
    println('I am the new one.')
}
```

执行 gradle -q copy

```
> gradle -q copy
I am the new one.
```

在这里我们用一个简单的任务替换 Copy 类型的任务。当创建这个简单的任务时，您必须将 overwrite 属性设置为 true。否则 Gradle 将抛出异常，说这种名称的任务已经存在。

15.8. Skipping tasks 跳过任务

Gradle 提供多种方式来跳过任务的执行。

15.8.1. Using a predicate 使用断言

你可以使用 onlyIf() 方法将断言附加到一项任务中。如果断言结果为 true，才会执行任务的操作。你可以用一个闭包来实现断言。闭包会作为一个参数传给任务，并且任务应该执行时返回 true，或任务应该跳过时返回false。断言只在任务要执行前才计算。

Example 15.20. Skipping a task using a predicate

build.gradle

```
task hello << {
    println 'hello world'
}

hello.onlyIf { !project.hasProperty('skipHello') }
```

执行 gradle hello -PskipHello

```
> gradle hello -PskipHello
:hello SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.8.2. Using StopExecutionException

如果跳过任务的规则不能与断言同时表达，您可以使用StopExecutionException。如果一个操作（action）抛出了此异常，那么这个操作（action）接下来的行为和这个任务的其他 操作（action）都会被跳过。构建会继续执行下一个任务。

Example 15.21. Skipping tasks with StopExecutionException

build.gradle

```
task compile << {
    println 'We are doing the compile.'
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}
task myTask(dependsOn: 'compile') << {
   println 'I am not affected'
}
```

Output of gradle -q myTask

```
> gradle -q myTask
I am not affected
```

如果您使用由 Gradle 提供的任务，那么此功能将非常有用。它允许您向一个任务的内置操作中添加执行条件。(你可能会想，为什么既不导入StopExecutionException 也没有通过其完全限定名来访问它。原因是，Gradle 会向您的脚本添加默认的一些导入。这些导入是可自定义的 （见Appendix E. Existing IDE Support and how to cope without it 支持的 IDE 以及如何应对没有它）。)

15.8.3. Enabling and disabling tasks 启用和禁用任务

每一项任务有一个默认值为 true 的 enabled 标记。将它设置为 false，可以不让这个任务的任何操作执行。

Example 15.22. Enabling and disabling tasks

build.gradle

```
task disableMe << {
    println 'This should not be printed if the task is disabled.'
}
disableMe.enabled = false
```

执行 gradle disableMe

```
> gradle disableMe
:disableMe SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.9. Skipping tasks that are up-to-date 跳过处于最新状态的任务

如果您使用 Gradle 自带的任务，如 Java 插件所添加的任务的话，你可能已经注意到 Gradle 将跳过处于最新状态的任务。这种行在您自己定义的任务上也有效，而不仅仅是内置任务。

15.9.1. Declaring a task's inputs and outputs 声明一个任务的输入和输出

让我们来看一个例子。在这里我们的任务从一个 XML 源文件生成多个输出文件。让我们运行它几次。

Example 15.23. A generator task

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

执行 gradle transform

```
> gradle transform
:transform
Transforming source file.
```

执行 gradle transform

```
> gradle transform
:transform
Transforming source file.
```

请注意 Gradle 第二次执行执行这项任务时，即使什么都未作改变，也没有跳过该任务。我们的示例任务被用一个操作（action）闭包来定义。Gradle 不知道这个闭包做了什么，也无法自动判断这个任务是否为最新状态。若要使用 Gradle 的最新状态（up-to-date）检查，您需要声明这个任务的输入和输出。

每个任务都有一个 inputs 和 outputs 的属性，用来声明任务的输入和输出。下面，我们修改了我们的示例，声明它将 XML 源文件作为输入，并产生输出到一个目标目录。让我们运行它几次。

Example 15.24. Declaring the inputs and outputs of a task

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    inputs.file srcFile
    outputs.dir destDir
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

执行 gradle transform

```
> gradle transform
:transform
Transforming source file.
```

执行 gradle transform

```
> gradle transform
:transform UP-TO-DATE
```

现在，Gradle 知道哪些文件要检查以确定任务是否为最新状态。

任务的 inputs 属性是 TaskInputs类型。任务的 outputs 属性是 TaskOutputs 类型。

一个没有定义输出的任务将永远不会被当作是最新的。对于任务的输出并不是文件的场景，或者是更复杂的场景， TaskOutputs.upToDateWhen())方法允许您以编程方式计算任务的输出是否应该被判断为最新状态。

一个只定义了输出的任务，如果自上一次构建以来它的输出没有改变，那么它会被判定为最新状态。

15.9.2. How does it work 它是怎么实现的？

在第一次执行任务之前，Gradle 对输入进行一次快照。这个快照包含了输入文件集和每个文件的内容的哈希值。然后 Gradle 执行该任务。如果任务成功完成，Gradle 将对输出进行一次快照。该快照包含输出文件集和每个文件的内容的哈希值。Gradle 会保存这两个快照，直到

任务的下一次执行。

之后每一次，在执行任务之前，Gradle 会对输入和输出进行一次新的快照。如果新的快照和前一次的快照一样，Gradle 会假定这些输出是最新状态的并跳过该任务。如果它们不一则，Gradle 则会执行该任务。Gradle 会保存这两个快照，直到任务的下一次执行。

请注意，如果一个任务有一个指定的输出目录，在它上一次执行之后添加到该目录的所有文件都将被忽略，并且不会使这个任务成为过时状态。这是不相关的任务可以在不互相干扰的情况下共用一个输出目录。如果你因为一些理由而不想这样，请考虑使用 TaskOutputs.upToDateWhen)

15.10. Task rules 任务规则

有时你想要有这样一项任务，它的行为依赖于参数数值范围的一个大数或是无限的数字。任务规则是提供此类任务的一个很好的表达方式：

Example 15.25. Task rule

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
```

执行 gradle -q pingServer1

```
> gradle -q pingServer1
Pinging: Server1
```

这个字符串参数被用作这条规则的描述。当对这个例子运行 gradle tasks 的时候，这个描述会被显示。

规则不只是从命令行调用任务才起作用。你也可以对基于规则的任务创建依赖关系

Example 15.26. Dependency on rule based tasks

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

执行 gradle -q groupPing

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

执行 gradle -q tasks 你找不到 “pingServer1” 或 “pingServer2” 任务，但脚本执行逻辑是基于请求执行这些任务的

15.11. Finalizer tasks 析构器任务

析构器任务是一个 孵化中 的功能 (请参阅Appendix C. The Feature Lifecycle 特性的生命周期 C.1.2 章节， “Incubating”)。

当最终的任务准备运行时，析构器任务会自动地添加到任务图中。

Example 15.27. Adding a task finalizer

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

执行 gradle -q taskX

```
> gradle -q taskX
taskX
taskY
```

即使最终的任务执行失败，析构器任务也会被执行。

Example 15.28. Task finalizer for a failing task

build.gradle

```
task taskX << {
    println 'taskX'
    throw new RuntimeException()
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

执行 gradle -q taskX

```
> gradle -q taskX
taskX
taskY
```

另一方面，如果最终的任务什么都不做的话，比如由于失败的任务依赖项或如果它被认为是最新的状态，析构任务不会执行。

在不管构建成功或是失败，都必须清理创建的资源的情况下，析构认为是很有用的。这样的资源的一个例子是，一个 web 容器会在集成测试任务前开始，并且在之后关闭，即使有些测试失败。

你可以使用 Task.finalizedBy()）方法指定一个析构器任务。这个方法接受一个任务实例、任务名称或Task.dependsOn())所接受的任何其他输入作为参数。

15.12. Summary 总结

如果你是从 Ant 转过来的，像 Copy 这种增强的 Gradle 任务，看起来就像是一个 Ant target（目标）和一个 Ant task (任务）之间的混合物。实际上确实是这样子。Gradle 没有像 Ant 那样对任务和目标进行分离。简单的 Gradle 任务就像 Ant 的目标，而增强的 Gradle 任

务还包括 Ant 任务方面的内容。Gradle 的所有任务共享一个公共 API，您可以创建它们之间的依赖性。这样的一个任务可能会比一个 Ant 任务更好配置。它充分利用了类型系统，更具有表现力而且易于维护。

Chapter 16. Working With Files 跟文件工作

大多数构建工作都要使用到文件。Gradle 添加了一些概念和 API 来帮助您实现这一目标。

16.1. Locating files 定位文件

使用 Project.file()) 方法来定位相对于 project 目录相关的文件。

Example 16.1. Locating files

build.gradle

```
// Using a relative path 使用相对路径
File configFile = file('src/config.xml')

// Using an absolute path 使用绝对路径
configFile = file(configFile.absolutePath)

// Using a File object with a relative path 使用文件对象中的相对路径
configFile = file(new File('src/config.xml'))
```

您可以把任何对象传递给 file() 方法，而它将尝试将其转换为一个绝对路径的 File 对象。通常情况下，你会传给它一个 String 或 File 的实例。如果这个路径是一个绝对路径，它会用于构构一个 File 实例。否则，会通过先计算所提供的路径相对于项目目录的相对路径来构造 File 实例。这个 file() 方法也可以识别 URL，例如是 file:/some/path.xml。

这是把一些用户提供的值转换为一个相对路径的 File 对象的有用方法。由于 file() 方法总是去计算所提供的路径相对于项目目录的路径，最好是使用 new File(somePath)，因为它是一个固定的路径，而不会因为用户运行 Gradle 的具体工作目录而改变。

16.2. File collections 文件集合

一个文件集合简单的说就是一组文件。它通过 FileCollection 接口来表示。Gradle API 中的许多对象都实现了此接口。比如，15.3 依赖配置章节 就实现了 FileCollection 这一接口。

使用 Project.files()) 方法是获取一个 FileCollection 实例的其中一个方法。你可以向这个方法传入任意个对象，而它们会被转换为一组 File 对象。这个 files() 方法接受任何类型的对象作为其参数。根据16.1 章节 “定位文件”里对 file() 方法的描述，它的结果会被计算为相对于项目目录的相对路径。你也可以将集合，迭代变量，map 和数组传递给 files() 方法。它们会被展开，并且内容会转换为 File 实例。

Example 16.2. Creating a file collection

build.gradle

```
FileCollection collection = files('src/file1.txt',
                                  new File('src/file2.txt'),
                                  ['src/file3.txt', 'src/file4.txt'])
```

一个文件集合是可迭代的，并且可以使用 as 操作符转换为其他类型的对象集合。您还可以使用 + 运算符把两个文件集合相加，或使用 - 运算符减去一个文件集合。这里是一些使用文件集合的例子

Example 16.3. Using a file collection

build.gradle

```
// Iterate over the files in the collection
collection.each {File file ->
    println file.name
}

// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile
File file2 = collection as File

// Add and subtract collections
def union = collection + files('src/file3.txt')
def different = collection - files('src/file3.txt')
```

你也可以向 files() 方法传一个闭包或一个 Callable 实例。它会在查询集合内容，并且它的返回值被转换为一组文件实例时被调用。这个闭包或Callable 实例的返回值可以是 files() 方法所支持的任何类型的对象。这是 “实现” FileCollection 接口的简单方法。

Example 16.4. Implementing a file collection

build.gradle

```
task list << {
    File srcDir

    // Create a file collection using a closure
    collection = files { srcDir.listFiles() }

    srcDir = file('src')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }

    srcDir = file('src2')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }
}
```

执行 gradle -q list

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

你可以向 files() 传入一些其他类型的对象：

FileCollection

它们会被展开，并且内容会被包含在文件集合内。

Task

任务的输出文件会被包含在文件集合内。

TaskOutputs

TaskOutputs 的输出文件会被包含在文件集合内。

要注意的一个地方是，一个文件集合的内容是缓计算的，它只在需要的时候才计算。这意味着您可以，比如创建一个 FileCollection 对象而里面的文件会在以后才创建，比方说在一些任务中才创建。

16.3. File trees 文件树

文件树是按层次结构排序的文件集合。例如，文件树可能表示一个目录树或 ZIP 文件的内容。它通过 FileTree 接口表示。FileTree 接口继承自FileCollection，所以你可以用对待文件集合一样的方式来对待文件树。Gradle 中的几个对象都实现了 FileTree 接口，例如 source sets。

使用 Project.fileTree() 方法是获取一个 FileTree 实例的其中一种方法。它将定义一个基目录创建 FileTree 对象，并可以选择加上一些 Ant 风格的包含与排除模式

Example 16.5. Creating a file tree

build.gradle

```
// Create a file tree with a base directory
FileTree tree = fileTree(dir: 'src/main')

// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// Create a tree using path
tree = fileTree('src').include('**/*.java')

// Create a tree using closure
tree = fileTree('src') {
    include '**/*.java'
}

// Create a tree using a map
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

你可以像使用一个文件集合的方式一样来使用一个文件树。你也可以使用 Ant 风格的模式来访问文件树的内容或选择一个子树：

Example 16.6. Using a file tree

build.gradle

```
// Iterate over the contents of a tree
tree.each {File file ->
    println file
}

// Filter a tree
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}

// Add trees together
FileTree sum = tree + fileTree(dir: 'src/test')

// Visit the elements of the tree
tree.visit {element ->
    println "$element.relativePath => $element.file"
}
```

16.4. Using the contents of an archive as a file tree 使用归档文件的内容作为文件树

您可以使用档案的内容，如 ZIP 或者 TAR 文件，作为一个文件树。你可以通过使用 Project.zipTree() 或 Project.tarTree() 方法来实现这一过程。这些方法返回一个 FileTree 实例，您可以像使用任何其他文件树或文件集合一样使用它。例如，您可以用它来通过复制内容扩大归档，或把一些档案合并到另一个归档文件中。

Example 16.7. Using an archive as a file tree

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')

// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')

//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

16.5. Specifying a set of input files 指定一组输入文件

Gradle 中的许多对象都有一个接受一组输入文件的属性。例如，JavaCompile 任务有一个 source 属性，定义了要编译的源代码文件。你可以使用上面所示的 files() 方法所支持的任意类型的对象设置此属性。这意味着您可以通过如 File、String、 集合、 FileCollection对象，或甚至是一个闭包来设置此属性。这里有一些例子：

Example 16.8. Specifying a set of files

build.gradle

```
// Use a File object to specify the source directory
compile {
    source = file('src/main/java')
}

// Use a String path to specify the source directory
compile {
    source = 'src/main/java'
}

// Use a collection to specify multiple source directories
compile {
    source = ['src/main/java', '../shared/java']
}

// Use a FileCollection (or FileTree in this case) to specify the source files
compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**' }
}

// Using a closure to specify the source files.
compile {
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect { zipTree(i
t) }
    }
}
```

通常情况下，有一个与属性相同名称的方法，可以追加这个文件集合。再者，这个方法接受 files() 方法所支持的任何类型的参数。

Example 16.9. Specifying a set of files

build.gradle

```
compile {
    // Add some source directories use String paths
    source 'src/main/java', 'src/main/groovy'

    // Add a source directory using a File object
    source file('../shared/java')

    // Add some source directories using a closure
    source { file('src/test/').listFiles() }
}
```

16.6. Copying files 拷贝文件

你可以使用 Copy 任务来复制文件。复制任务非常灵活，并允许您进行，比如筛选要复制的文件的内容，或映射文件的名称。

若要使用 Copy 任务，您必须提供用于复制的源文件和目标目录。您还可以在复制文件的时候指定如何转换文件。你可以使用一个复制规范来做这些。一个复制规范通过 CopySpec 接口来表示。Copy 任务实现了此接口。你可以使用 CopySpec.from() 方法指定源文件，使用 CopySpec.into() 方法使用目标目录。

Example 16.10. Copying files using the copy task

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
}
```

from() 方法接受和 files() 方法一样的任何参数。当参数解析为一个目录时，该目录下的所有文件（不包含目录本身） 都会递归复制到目标目录。当参数解析为一个文件时，该文件会复制到目标目录中。当参数解析为一个不存在的文件时，参数会被忽略。如果参数是一个任务，那么任务的输出文件 （即该任务创建的文件）会被复制，并且该任务会自动添加为Copy 任务的依赖项。 into() 方法接受和 files() 方法一样的任何参数。这里是另一个示例：

Example 16.11. Specifying copy task source files and destination directory

build.gradle

```
task anotherCopyTask(type: Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
    into { getDestDir() }
}
```

您可以使用 Ant 风格的包含或排除模式，或使用一个闭包，来选择要复制的文件：

Example 16.12. Selecting the files to copy

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') &&
                         details.file.text.contains('staging') }
}
```

此外，你也可以使用 Project.copy() 方法来复制文件。它是与任务一样的工作方式，尽管它有一些主要的限制。首先， copy() 不能进行增量操作（见15.9章节，"跳过处于最新状态的任务"）。

Example 16.13. Copying files using the copy() method without up-to-date check

build.gradle

```
task copyMethod << {
    copy {
        from 'src/main/webapp'
        into 'build/explodedWar'
        include '**/*.html'
        include '**/*.jsp'
    }
}
```

第二，当一个任务用作复制源（即作为 from() 的参数）的时候，copy()方法不能建立任务依赖性，因为它是一个方法，而不是一个任务。因此，如果您在任务的 action 里面使用 copy() 方法，必须显式声明所有的输入和输出以得到正确的行为。

Example 16.14. Copying files using the copy() method with up-to-date check

build.gradle

```
task copyMethodWithExplicitDependencies{
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.file copyTask
    outputs.dir 'some-dir' // up-to-date check for outputs
    doLast{
        copy {
            // Copy the output of copyTask
            from copyTask
            into 'some-dir'
        }
    }
}
```

在可能的情况下，最好是使用 Copy 任务，因为它支持增量构建和任务依赖关系推理，而不需要你额外付出。copy() 方法可以作为一个任务执行的部分来复制文件。即，这个 copy() 方法旨在用于自定义任务 （见Chapter 58. Writing Custom Task Classes 编写自定义任务类）中，需要文件复制作为其一部分功能的时候。在这种情况下，自定义任务应充分声明与复制操作有关的输入/输出。

16.6.1. Renaming files 重命名

Example 16.15. Renaming files as they are copied

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Use a closure to map the file name
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // Use a regular expression to map the file name
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

16.6.2. Filtering files 过滤文件

Example 16.16. Filtering files as they are copied

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
}
```

在源文件中，“expand” 和 “filter” 操作查找的 “token” ，被格式化成“@tokenName@” （名称是 “tokenName”）

16.6.3. Using the CopySpec class 使用CopySpec类

复制规范用来组织一个层次结构。一个复制规范继承其目标路径，包含模式，排除模式，复制操作，名称映射和过滤器。

Example 16.17. Nested copy specs

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

16.7. Using the Sync task 使用Sync任务

Sync 任务继承了 Copy 任务。当它执行时，它会将源文件复制到目标目录中，然后从目标目录移除所有不是它复制的文件。这可以用来做一些事情，比如安装你的应用程序、 创建你的归档文件的 exploded 副本，或维护项目的依赖项的副本。

这里是一个例子，维护在 build/libs 目录中的项目运行时依赖的副本。

Example 16.18. Using the Sync task to copy dependencies

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

16.8. Creating archives 创建归档文件

一个项目可以有你所想要的一样多的 JAR 文件。您也可以将 WAR、 ZIP 和TAG 文件添加到您的项目。使用各种归档任务可以创建以下的归档文件： Zip, Tar, Jar, War, and Ear. 他们的工作方式都一样，所以让我们看看如何创建一个 ZIP 文件。

Example 16.19. Creating a ZIP archive

build.gradle

```
apply plugin: 'java'

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
}
```

归档任务与 Copy 任务的工作方式一样，并且实现了相同的 CopySpec 接口。像使用 Copy 任务一样，你需要使用 from() 的方法指定输入的文件，并可以选择是否通过 into() 方法指定最终在存档中的位置。您可以通过一个复制规范来筛选文件的内容、 重命名文件和进行其他你可以做的事情。

16.8.1. Archive naming

生成的归档的默认名称是 projectName-version.type。举个例子：

Example 16.20. Creation of ZIP archive

build.gradle

```
apply plugin: 'java'

version = 1.0

task myZip(type: Zip) {
    from 'somedir'
}

println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

执行 gradle -q myZip

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-1.0.zip
```

它添加了一个名称为 myZip 的 ZIP 归档任务，产生 ZIP 文件 zipProject 1.0.zip 。区分归档任务的名称和归档任务生成的归档文件的名称是很重要的。归档的默认名称可以通过项目属性 archivesBaseName 来更改。还可以在以后的任何时候更改归档文件的名称。

这里有很多你可以在归档任务中设置的属性。它们在以下的表 16.1，"存档任务-命名属性"中列出。你可以，比方说，更改归档文件的名称：

Example 16.21. Configuration of archive task - custom archive name

build.gradle

```
apply plugin: 'java'
version = 1.0

task myZip(type: Zip) {
    from 'somedir'
    baseName = 'customName'
}

println myZip.archiveName
```

执行 gradle -q myZip

```
> gradle -q myZip
customName-1.0.zip
```

您可以进一步自定义存档名称：

Example 16.22. Configuration of archive task - appendix & classifier

build.gradle

```
apply plugin: 'java'
archivesBaseName = 'gradle'
version = 1.0

task myZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    from 'somedir'
}

println myZip.archiveName
```

执行 gradle -q myZip

```
> gradle -q myZip
gradle-wrapper-1.0-src.zip
```

Table 16.1. Archive tasks - naming properties

属性名称	类型	默认值	描述
`archiveName`	`String`	baseName-appendix-version-classifier.extension 如果这些属性中的任何一个为空，那后面的-不会被添加到该名称中。	生成的归档文件的基本文件名
`archivePath`	`File`	`destinationDir/archiveName`	生成的归档文件的绝对路径。
`destinationDir`	`File`	依赖于归档类型。JAR包和 WAR包会生成到 project.buildDir/libraries中。ZIP文件和TAR文件会生成到project.buildDir/distributions中。	存放生成的归档文件的目录
`baseName`	`String`	`project.name`	归档文件的名称中的基本名称部分。
`appendix`	`String`	`null`	归档文件的名称中的附录部分
`version`	`String`	`project.version`	归档文件的名称中的版本部分。
`classifier`	`String`	`null`	归档文件的名称中的分类部分。
`extension`	`String`	依赖于归档的类型，用于TAR文件，可以是以下压缩类型： zip, jar, war, tar, tgz or tbz2.	归档文件的名称中的扩展名称部分。

16.8.2. Sharing content between multiple archives 共享多个归档之间的内容

你可以使用Project.copySpec())方法在归档之间共享内容。

你经常会想要发布一个归档文件，这样就可从另一个项目中使用它。这一过程在Chapter 52. Publishing artifacts 发布 artifact会讲到

Chapter 17. Using Ant from Gradle 从 Gradle 使用 Ant

Gradle 提供了对 Ant 的优秀集成。可以在你的 Gradle 构建中，使用单独的 Ant 任务或整个 Ant 构建。事实上，你会发现在 Gradle 中使用 Ant 任务比使用 Ant 的 XML 格式更容易也更强大。你甚至可以只把 Gradle 当作一个强大的 Ant 任务脚本的工具。

Ant 可以分为两层。第一层是 Ant 的语言。它提供了用于 build.xml，处理的目标，特殊的构造方法比如宏，还有其他等等的语法。换句话说，除了 Ant 任务和类型之外全部都有。Gradle 理解这种语言，并允许您直接导入你的 Ant build.xml 到 Gradle 项目中。然后你可以使用你的 Ant 构建中的 target，就好像它们是 Gradle task 一样。

Ant 的第二层是其丰富的 Ant 任务和类型，如 javac、copy或jar。这一层 Gradle 只靠 Groovy 和非常棒的 AntBuilder，对其提供了集成。

最后，由于构建脚本是 Groovy 脚本，所以您始终可以作为一个外部进程来执行 Ant 构建。你的构建脚本可能包含有类似这样的语句：`"ant clean compile".execute()` (在 Groovy 中，你可以执行字符串。了解更多关于执行外部进程，请查看 9.3.2 的'Groovy in Action'或者 Groovy wiki)

你可以把 Gradle 的 Ant 集成当成一个路径，将你的构建从 Ant 迁移至 Gradle 。例如，你可以通过导入您现有的 Ant 构建来开始。然后，可以将您的依赖声明从 Ant 脚本移到您的构建文件。最后，您可以将整个任务移动到您的构建文件，或者把它们替换为一些 Gradle 插件。这个过程可以随着时间一点点完成，并且在这整个过程当中你的 Gradle 构建都可以使用用。

17.1. Using Ant tasks and types in your build 在你的构建中使用 Ant 的任务和类型

在构建脚本中，Gradle 提供了一个名为 ant 的属性。它指向一个 AntBuilder 实例。AntBuilder 用于从你的构建脚本中访问 Ant 任务、 类型和属性。从 Ant 的 build.xml 格式到 Groovy 之间有一个非常简单的映射，下面解释。

通过调用 AntBuilder 实例上的一个方法，可以执行一个 Ant 任务。你可以把任务名称当作方法名称使用。例如，你可以通过调用 ant.echo() 方法执行 Ant 的 echo 任务。Ant 任务的属性会作为 Map 参数传给该方法。下面是执行 echo 任务的例子。请注意我们还可以混合使用 Groovy 代码和 Ant 任务标记。这将会非常强大。

Example 17.1. Using an Ant task

build.gradle

```
task hello << {
    String greeting = 'hello from Ant'
    ant.echo(message: greeting)
}
```

执行 gradle hello

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

你可以把一个嵌套文本，通过作为任务方法调用的参数，把它传给一个 Ant 任务。在此示例中，我们将把作为嵌套文本的消息传给 echo 任务：

Example 17.2. Passing nested text to an Ant task

build.gradle

```
task hello << {
    ant.echo('hello from Ant')
}
```

执行 gradle hello

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

你可以在一个闭包里把嵌套的元素传给一个 Ant 任务。嵌套元素的定义方式与任务相同，通过调用与我们要定义的元素一样的名字的方法

Example 17.3. Passing nested elements to an Ant task

build.gradle

```
task zip << {
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'src') {
            include(name: '**.xml')
            exclude(name: '**.java')
        }
    }
}
```

您可以用访问任务同样的方法，把类型名字作为方法名称，访问 Ant 类型。方法调用返回 Ant 数据类型，然后可以在构建脚本中直接使用。在以下示例中，我们创建一个 Ant 的 path 对象，然后循环访问它的内容。

Example 17.4. Using an Ant type

build.gradle

```
task list << {
    def path = ant.path {
        fileset(dir: 'libs', includes: '*.jar')
    }
    path.list().each {
        println it
    }
}
```

更多有关 AntBuilder 见 8.4 的'Groovy in Action' 或者 Groovy Wiki

17.1.1. Using custom Ant tasks in your build 在构建中使用自定义的 Ant 任务

要使自定义任务在您的构建中可用，你可以使用 Ant 任务 taskdef（通常更容易） 或 typedef，就像在 build.xml 文件中一样。然后，您可以像引用内置的 Ant 任务一样引用自定义 Ant 任务。

Example 17.5. Using a custom Ant task

build.gradle

```
task check << {
    ant.taskdef(resource: 'checkstyletask.properties') {
        classpath {
            fileset(dir: 'libs', includes: '*.jar')
        }
    }
    ant.checkstyle(config: 'checkstyle.xml') {
        fileset(dir: 'src')
    }
}
```

你可以使用 Gradle 的依赖管理组合类路径，以用于自定义任务。要做到这一点，你需要定义一个自定义配置的类路径中，然后将一些依赖项添加到配置中。这在 50.4章节，“如何声明你的依赖”有更详细的描述。

Example 17.6. Declaring the classpath for a custom Ant task

build.gradle

```
configurations {
    pmd
}

dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

若要使用类路径配置，请使用自定义配置里的 asPath 属性。

Example 17.7. Using a custom Ant task and dependency management together

build.gradle

task check << { ant.taskdef(name: 'pmd', classname: 'net.sourceforge.pmd.ant.PMDTask', classpath: configurations.pmd.asPath) ant.pmd(shortFilenames: 'true', failonruleviolation: 'true', rulesetfiles: file('pmd-rules.xml').toURI().toString()) { formatter(type: 'text', toConsole: 'true') fileset(dir: 'src') } }

17.2. Importing an Ant build 导入 Ant 的构建

你可以使用 ant.importBuild() 方法来向 Gradle 项目导入一个 Ant 构建。当您导入一个 Ant 构建时，每个 Ant 目标被视为一个 Gradle 任务。这意味着你可以用与 Gradle 任务完全相机的方式操纵和执行 Ant 目标。

Example 17.8. Importing an Ant build

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

执行 gradle hello

```
> gradle hello
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

您可以添加一个依赖于 Ant 目标的任务：

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello) << {
    println 'Hello, from Gradle'
}
```

执行 gradle intro

```
> gradle intro
:hello
[ant:echo] Hello, from Ant
:intro
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

或者，您可以将行为添加到 Ant 目标中：

Example 17.10. Adding behaviour to an Ant target

build.gradle

```
ant.importBuild 'build.xml'

hello << {
    println 'Hello, from Gradle'
}
```

执行 gradle hello

```
> gradle hello
:hello
[ant:echo] Hello, from Ant
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

它也可以用于一个依赖于 Gradle 任务的 Ant 目标：

Example 17.11. Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

task intro << {
    println 'Hello, from Gradle'
}
```

build.xml

```
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

执行 gradle hello

```
> gradle hello
:intro
Hello, from Gradle
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs‘
```

有时可能需要“改名”来避免 Ant target 生成的 task 与现有的 Gradle task 在命名上冲突。使用 AntBuilder.importBuild()) 方法

Example 17.12. Renaming imported Ant targets

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

执行 gradle a-hello

```
> gradle a-hello
:a-hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

请注意，虽然这个方法的第二个参数应该是一个 Transformer，在 Groovy 编程时可以使用简单的闭包而不是一个匿名内部类（或类似的），因为 Groovy 的支持自动强制关闭 single-abstract-method（单一的抽象方法）的类型。

17.3. Ant properties and references 关于 Ant 的属性和引用

有几种方法来设置 Ant 属性，以便使该属性被 Ant 任务使用。你可以直接在 AntBuilder 实例上设置属性。Ant 属性也可以从一个你可以修改的 Map 中获得。您还可以使用 Ant property 任务。下面是一些如何做到这一点的例子。

Example 17.13. Setting an Ant property

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

build.xml

```
<echo>buildDir = ${buildDir}</echo>
```

有几种方法可以设置 Ant 引用：

Example 17.14. Getting an Ant property

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

有几种方法可以获取 Ant 引用：

Example 17.15. Setting an Ant reference

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.xml

```
<path refid="classpath"/>
```

有几种方法可以获取 Ant 引用：

Example 17.16. Getting an Ant reference

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle

```
println ant.references.antPath
println ant.references['antPath']
```

17.4. API

AntBuilder 提供 Ant 的集成

Chapter 18. Logging 日志

日志是构建工具的主要"UI"。如果日志太多，真正的警告和问题容易被隐藏。另一方面，如果出了错，你需要找出相关的信息。Gradle 定义了6个日志级别，如表 Table 18.1, “Log levels” 所示。除了那些您通过可能会平常看到的日志级别之外，有两个 Gradle 特定日志级别。这两个级别分别是 `QUIET` 和 `LIFECYCLE` 。默认使用后面的这个日志级别，用于报告构建进度。

Table 18.1. Log levels

级别	用途
ERROR	Error 错误信息
QUIET	重要信息
WARNING	Warning 警告信息
LIFECYCLE	过程信息
INFO	信息
DEBUG	Debug 调试信息

18.1. Choosing a log level 选择级别

在 Table 18.2, “Log level command-line options” 中命令行，是用来选择不同的级别的选项。Table 18.3, “Stacktrace command-line options” 中的是影响堆栈跟踪日志

Table 18.2. Log level command-line options

选项	输出日志的级别
no logging options	LIFECYCLE 及更高
`-q` or `--quiet`	QUIET 及更高
`-i` or `--info`	INFO 及更高
`-d` or `--debug`	DEBUG 及更高 (所有的日志信息)

Table 18.3. Stacktrace command-line options

选项	含义
No stacktrace options	构建错误（如编译错误）时没有栈跟踪打印到控制台。只有在内部异常的情况下才打印栈跟踪。如果选择 DEBUG 日志级别，则总是输出截取后的栈跟踪信息。
`-s` or `--stacktrace`	输出截断的栈跟踪。我们推荐使用这一个选项而不是打印全栈的跟踪信息。Groovy 的全栈跟踪非常冗长 （由于其潜在的动态调用机制，然而他们通常不包含你的的代码中哪里错了的相关信息。）
`-S` or `--full-stacktrace`	打印全栈的跟踪信息。

18.2. Writing your own log messages 编写自己的日志消息

在构建文件，打印日志的一个简单方法是把消息写到标准输出中。Gradle 会把写到标准输出的所有内容重定向到它的日志系统的 QUIET 级别中。

Example 18.1. Using stdout to write log messages

build.gradle

```
println 'A message which is logged at QUIET level'
```

Gradle 还提供了一个 logger 属性给构建脚本，它是一个 Logger 实例。该接口扩展自 SLF4J 的 Logger接口，并添加了几个 Gradle 的特有方法。下面是关于如何在构建脚本中使用它的示例：

Example 18.2. Writing your own log messages

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

您也可以在构建脚本中通过其他使用的类挂钩到 Gradle 的日志系统中（例如 buildSrc 目录中的类）。只需使用一个 SLF4J 的logger对象。你可以在构建脚本中，用与内置的logger同样的方式使用这个logger。

Example 18.3. Using SLF4J to write log messages

build.gradle

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory

Logger slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

18.3. Logging from external tools and libraries 使用外部工具和库记录日志

Gradle 内部使用 Ant 和 Ivy。它们都有自己的日志系统。Gradle 将他们日志输出重定向到 Gradle 的日志系统。从 Ant/Ivy 的日志级别到 Gradle 的日志级别是一对一的映射，除了 Ant/Ivy 的 TRACE 级别，它是映射到 Gradle 的 DEBUG 级别的。这意味着默认情况下，Gradle 日志级别将不会显示任何 Ant/Ivy 的输出，除非是错误或警告信息。

有很多的工具仍然在使用标准输出日志记录。默认情况下，Gradle 将标准输出重定向到 QUIET日志级别，把标准错误输出重写向到 ERROR 级别。这种行为是可配置的。Project 对象提供了一个 LoggingManager，它允许您在计算构建脚本时，修改标准输出和错误重定向的日志级别。

Example 18.4. Configuring standard output capture

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

为能在任务执行过程中更改标准输出或错误的日志级别，task也提供了一个 LoggingManager。

Example 18.5. Configuring standard output capture for a task

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

Gradle 还提供了对 Java Util Logging，Jakarta Commons Logging 和 Log4j 的日志工具的集成。你生成的类使用这些日志记录工具输出的任何日志消息，都将被重定向到 Gradle 的日志系统。

18.4. Changing what Gradle logs 改变 Gradle 日志

您可以用您自己的 logging UI 大量地替换 Gradle 的。你可以这样做，例如，如果您想要以某种方式自定义 UI ——以输出更多或更少的信息，或修改日志格式您可以使用 Gradle.useLogger() 方法替换这个 logging。它可以在构建脚本，或 init 脚本，或通过内嵌的 API 访问。请注意它完全禁用 Gradle 的默认输出。下面是一个示例，在 init 脚本中修改任务执行和构建完成的日志打印。

Example 18.6. Customizing what Gradle logs

init.gradle

```
useLogger(new CustomEventLogger())

class CustomEventLogger extends BuildAdapter implements TaskExecutionListener {

    public void beforeExecute(Task task) {
        println "[$task.name]"
    }

    public void afterExecute(Task task, TaskState state) {
        println()
    }

    public void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

执行 gradle -I init.gradle build

```
> gradle -I init.gradle build
[compile]
compiling source

[testCompile]
compiling test source

[test]
running unit tests

[build]

build completed
```

你的 logger 可以实现下面列出的任何监听器接口。当你注册一个 logger时，只能替换它实现的接口的日志记录。其他接口的日志记录是不变的。你可以在 The Build Lifecycle 构建生命周期中的 55.6 节 “在构建脚本中响应生命周期”查看相关信息。

- BuildListener
- ProjectEvaluationListener
- TaskExecutionGraphListener
- TaskExecutionListener
- TaskActionListener

Chapter 19. The Gradle Daemon 守护进程

19.1. Enter the daemon 走进守护进程

Gradle 守护进程（有时也称为构建守护进程） 的目的是改善 Gradle 的启动和执行时间。

我们准备了几个守护进程非常有用的用例。对于一些工作流，用户会多次调用 Gradle，以执行少量的相对快速的任务。举个例子：

- 当使用测试驱动开发时，单元测试会被执行多次。
- 当开发一个 web 应用程序中，应用程序会被组装多次。
- 当发现构建能做什么，在 gradle tasks 在哪里会执行多次。

对以上各种工作流来说，让调用 Gradle 的启动成本尽可能小会很重要。

此外，如果可以相对较快地建立 Gradle 模型，用户界面可以提供一些有趣的功能。例如，该守护进程可能用于以下情形：

- 在 IDE 中的内容帮助
- 在 GUI 中的实时可视化构建
- 在 CLI 中的 tab 键完成

一般情况下，构建工具的敏捷行为总是可以派上用场。如果你尝试在你的本地构建中使用守护进程的话，它会变得让你很难回到正常的 Gradle 使用。

Tooling API (参见 Chapter 63. Embedding Gradle 嵌入 Gradle) 在整个过程当中都使用守护进程。如，你无法在没有守护进程时正式地使用 Tooling API。这意味着当您在 Eclipse中使用 STS Gradle 或在 Intellij IDEA 中使用 Gradle 支持时，您已经在使用 Gradle 守护进程。

未来，该守护进程还会提供更多的功能：

- 敏捷的 up-to-date 检查：使用本地文件系统修改通知（例如，通过 jdk7 nio.2）预先执行 up-to-date 分析。
- 更快的构建： 预评估项目，这样当用户接下来调用 Gradle 时，模型就准备好了。
- 我们提到了更快的构建吗？守护进程可以预先下载依赖项或进行快照依赖的新版本检查。
- 使用可用于编译和测试的一个可复用线程池。例如，Groovy 和 Scala 的编译器启动开销都很大。构建守护进程可以维持一个已下载的 Groovy 和 （或） Scala 进程。
- 预先执行某些任务，比如编译。更快的反馈。
- 快速、 准确的 bash 的 tab 键完成。
- Gradle 缓存的定期垃圾收集。

19.2. Reusing and expiration of daemons 重用和失效的守护程序

基本的思想是， gradle 命令会 fork 一个守护进程，用于执行实际的构建。Gradle 命令的后续调用将重用该守护进程，以避免启动开销。有时我们不能使用现有的守护进程，是因为它正忙或其 java 版本或 jvm 参数不同。关于fork一个完全新的守护进程的具体细节，请阅读下面的专题。守护进程将在空闲3小时后自动失效。

以下是我们 fork 一个新的守护进程的所有情况：

- 如果该守护进程当前正忙于运行一些作业，将启动一个全新的守护进程。 对每个java home，我们会fork一个单独的守护进程。所以即使有一些闲置的守护进程等待构建请求，但你碰巧通过不同的 java HOME 运行构建，那么一个全新的守护进程将会被 fork。
- 如果用于构建的 jvm 的参数足够不同，我们会 fork 一个单独的守护进程。例如，如果某些系统属性已经更改，我们不会 fork 一个新的守护进程。然而，如果 -Xmx 内存设置更改了，或一些基本的不变的系统属性更改了 （例如 file.encoding），那么将 fork 新的守护进程。
- 在这一刻，守护进程会被加上 Gradle 的特定版本号。这意味着即使一些守护进程处于空闲状态，但您正在运行的构建与 Gradle 不同版本，也将启动一个新的守护进程。这也有一种 --stop 命令行指令的结果： 当运行 --stop 时，您仅可以停止以你的 Gradle 版本启动的守护进程。

我们计划在将来改进守护进程的managing / pooling的方法。

19.3. Usage and troubleshooting 用法和故障排除

关于命令行的用法，可以看一下专题Appendix D. Gradle Command Line 命令行。如果你已经厌倦反复使用相同的命令行选项，可以看看第 20.1 章节，“通过 gradle.properties 配置构建环境”。这一章节包含了有关如何以一种“持久化”的方式配置某些行为（包括在默认情况下打开守护进程）的信息。

以下是有关 Gradle 守护进程的故障排除的一些方面：

- 如果你的构建有问题，请尝试暂时禁用守护进程 （您可以通过使用命令行开关--no-daemon)。
- 有时候，您可能想要通过--stop命令行选项或更有力的方式停止守护程序。
- 默认情况下位于 Gradle 用户主目录有一个守护进程的日志文件。
- 你可能想要以--foreground模式启动守护程序序，以观察构建是怎么执行的。

19.4. Configuring the daemon 配置守护进程

可以配置一些守护进程的设置，例如 JVM 参数、 内存设置或Java home目录。有关更多信息请参阅20.1章节，“通过 gradle.properties 配置构建环境”

Chapter 20. The Build Environment 构建环境

20.1. 通过 gradle.propert 配置构建环境

Gradle 提供几个选项,使它容易配置将用于执行构建的 Java 进程。同时可以通过 GRADLE_OPTS 或 JAVA_OPTS 配置这些在你本地环境,包含的设置包括比如 JVM 内存设置,Java home,守护进程开/关,它们可以和你的项目在你的版本控制系统中被版本化的话，将会更有用，这样整个团队就可以使用一致的环境了。在你的构建当中，建立一致的环境，就和把这些配置放进 gradle.properties 文件一样简单。这些配置将会按以下顺序被应用（以防在多个地方都有配置时只有最后一个 生效）

- 从 gradle.properties 在项目构建 dir。
- 从 gradle.properties 在 gradle user home.
- 从系统属性,例如当 -Dsome.property 在命令行上设置。

可以使用以下属性来配置 Gradle 构建环境:

org.gradle.daemon

当设置为true 时，Gradle 守护进程会运行构建。对于本地开发者的构建而言，这是我们最喜欢的属性。开发人员的环境在速度和反馈上会优化，所以我们几乎总是使用守护进程运行 Gradle 作业。由于 CI 环境在一致性和可靠性上的优化，我们不通过守护进程运行 CI 构建（即长时间运行进程）

org.gradle.java.home

为 Gradle 构建进程指定 java home 目录。这个值可以设置为 jdk 或jre 的位置，不过，根据你的构建所做的，选择 jdk 会更安全。如果该设置未指定，将使用合理的默认值。

org.gradle.jvmargs

指定用于该守护进程的 jvmargs。该设置对调整内存设置特别有用。目前的内存上的默认设置很大方。

org.gradle.configureondemand

启用新的孵化模式，可以在配置项目时使得 Gradle 具有选择性。只适用于相关的项目被配置为在大型多项目中更快地构建。请参阅 Section 57.1.1.1, “Configuration on demand”.

org.gradle.parallel

如果配置了这一个，Gradle 将在孵化的并行模式下运行。

20.1.1. Forked Java processes

许多设置（如 Java 版本和最大堆大小）只能在启动一个新的 JVM 构建进程时指定。这意味着 Gradle 在分析了各种 gradle.properties 文件之后，必须启动一个单独的 JVM 进程，以执行构建操作。当通过守护进程运行时，带有正确参数的 JVM 会启动一次，并在每次的守护进程构建执行时复用。当不通过守护进程执行 Gradle 时，在每次构建执行中都必须启动一个新的 JVM ，除非 JVM 是由 Gradle 启动脚本启动的，并且恰好具有相同的参数。

在执行每个构建时运行一个额外的 JVM 的代价是非常昂贵的，这就是为什么我们强烈推荐您使用 Gradle 守护进程，如果你指定了org.gradle.java.home 或 org.gradle.jvmargs。更多详细信息，请参阅Chapter 19. The Gradle Daemon 守护进程。

20.2. Accessing the web via a proxy 通过代理访问 web

配置 HTTP 代理服务器 （例如用于下载依赖） 是通过标准的 JVM 系统属性来做的。这些属性可以直接在构建脚本中设置；例如设置代理主机为System.setProperty （'http.proxyHost', 'www.somehost.org'）。或者，可以在构建的根目录或 Gradle 主目录中的 gradle.properties 文件中指定这些属性。

xample 20.1. Configuring an HTTP proxy

gradle.properties

```
systemProp.http.proxyHost=www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

对于 HTTPS 有单独的设置。

Example 20.2. Configuring an HTTPS proxy

gradle.properties

```
systemProp.https.proxyHost=www.somehost.org
systemProp.https.proxyPort=8080
systemProp.https.proxyUser=userid
systemProp.https.proxyPassword=password
systemProp.https.nonProxyHosts=*.nonproxyrepos.com|localhost
```

我们无法很好地概述所有可能的代理服务器设置。其中可以去看的一个地方是 Ant 项目的一个文件中的常量。这里是Subversion 的视图的链接。另一个地方是 JDK 文档的Networking Properties(网络属性)。如果有人知道更好的概述，请发邮件让我们知道。

20.2.1. NTLM Authentication

如果您的代理服务器需要 NTLM 身份验证，您可能需要提供验证域，以及用户名和密码。有两种方法可以向 NTLM 代理提供验证域：

- 将 http.proxyUser 系统属性设置为一个这样的值：域/用户名。
- 通过 http.auth.ntlm.domain 系统属性提供验证域。

Chapter 21. Gradle Plugins 插件

Gradle 在它的核心中有意地提供了一些小但有用的功能，用于在真实世界中的自动化。所有有用的功能，例如以能够编译 Java 代码为例，都是通过插件进行添加的。插件添加了新任务（例如 JavaCompile），域对象 （例如 SourceSet ），约定（例如主要的 Java 源代码是位于 src/main/java ），以及扩展的核心对象和其他插件的对象。

在这一章中，我们将讨论如何使用插件以及术语和插件相关的概念。

21.1. Types of plugins 插件类型

Gradle 一般有两种类型的插件 :script 插件和 binary 插件。script 插件是额外的构建脚本，用于进一步配置构建，以及实现一种声明性方法操纵构建。他们通常用于构建虽然他们可以外部化和从远程位置访问。binary插件的类实现 Plugin 接口,采用的编程方法操纵构建。binary 插件通过项目层次结构或外部插件 jar 驻留在构建脚本中。

21.2. Applying plugins 应用插件

插件是可以被应用的，通过 Project.apply()) 方法来完成。

21.2.1. Script plugins

Example 21.1. Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

script 插件可以从本地文件系统或在远程位置的脚本应用。文件位置是相对于项目目录，而远程脚本的位置是一个 HTTP URL 指定。多个脚本插件（或形式）可以应用到一个给定的建立。

21.2.2. Binary plugins

Example 21.2. Applying a binary plugin

build.gradle

```
apply plugin: 'java'
```

核心插件注册一个简短的名字。在上面的例子中，我们使用短名称“java”来应用 JavaPlugin。插件也有插件ID，以一个完全合格的形式如 com.github.foo.bar，虽然一些遗留的插件还可以利用短期的，不合格的形式。

该方法还可以接受一个类识别插件：

Example 21.3. Applying a binary plugin by type

build.gradle

```
apply plugin: JavaPlugin
```

在上述样本JavaPlugin 符号就是指 JavaPlugin。这类不需要严格引入 org.gradle.api.plugins 包在所有构建脚本会自动导入（见Appendix E. Existing IDE Support and how to cope without it 支持的 IDE 以及如何应对没有它）。此外，不需要追加 .class 来确认这个类是在 Groovy 还是在 Java。

插件的应用是幂等。就是说一个插件，可多次应用。如果插件已被应用，任何进一步的应用将没有任何效果。

21.2.2.1. Locations of binary plugins 关于 binary 插件的位置

一个插件是任意类实现 Plugin 接口。Gradle 提供核心插件为其分配部分简单地应用插件来提供所有你需要做的。然而，非核心 binary 插件需要可用于构建类路径才可以应用。这可以通过很多途径实现，包括：

- 定义插件为内联类的声明在一个构建脚本。
- 定义插件作为一个源在项目目录下的文件 buildsrc。
- 包括从外部罐定义为 buildscript 依赖插件（见60.5节，“外部依赖关系的构建脚本”）。
- 包括插件从插件门户使用插件的DSL（见21.3节，“应用插件与插件DSL”）。

更多关于定义你自己的插件，见Chapter 59. Writing Custom Plugins 编写自定义插件。

21.3. Applying plugins with the plugins DSL 使用DSL应用插件

插件DSL目前正在酝酿中。请注意,DSL和其他配置可能会在以后 Gradle版本改变。

新的插件提供了一个更简洁的 DSL 和方便的方式来声明插件的依赖关系。它的工作原理与新的 Gradle 插件门户 提供方便地访问核心和社区插件。插件脚本块配置实例 PluginDependenciesSpec。

申请一个核心插件，可以使用短名称：

Example 21.4. Applying a core plugin

build.gradle

```
plugins {
    id 'java'
}
```

通过To apply a community plugin from the portal, the fully qualified plugin id must be used:

Example 21.5. Applying a community plugin

build.gradle

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

不需要进一步配置。具体来说,不需要配置 buildscript 类路径。Gradle将在插件门户解决插件,找到它,使它可用于构建。

有关更多信息,请参见 PluginDependenciesSpec 使用插件的 DSL。

21.4. Finding community plugins 寻找社区插件

Gradle 至今有一个充满活力的社区插件开发人员贡献为各种功能的插件。Gradle 插件门户提供了一个接口用于搜索和探索社区插件。

21.5. What plugins do 插件做啥

把插件应用到项目中可以让插件来扩展项目的功能。它可以做的事情如：

- 将任务添加到项目 （如编译、 测试）
- 使用有用的默认设置对已添加的任务进行预配置。
- 向项目中添加依赖配置 （见第 8.3 节，“依赖配置”） 。
- 通过扩展对现有类型添加新的属性和方法。

举例

Example 21.6. Tasks added by a plugin

build.gradle

```
apply plugin: 'java'

task show << {
    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

执行 gradle -q show

```
> gradle -q show
build/classes/main
build/resources/main
```

这个 Java 插件增加了 compileJava 任务 processResources 任务到项目中，以及给这两个任务配置 destinationDir 属性。

21.6. Conventions 约定

插件可以通过智能的方法对项目进行预配置以支持约定优于配置。Gradle 对此提供了机制和完善的支持，而它是强大-然而-简洁的构建脚本中的一个关键因素。

在上面的示例中我们看到，Java 插件添加了一个任务，名字为compileJava ，有一个名为 destinationDir 的属性（即配置编译的 Java 代码存放的地方）。Java 插件默认此属性指向项目目录中的build/classes/main。这是通过一个合理的默认的约定优于配置的例子。

我们可以简单地通过给它一个新的值来更改此属性。

Example 21.7. Changing plugin defaults

build.gradle

```
apply plugin: 'java'

compileJava.destinationDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

执行 gradle -q show

```
> gradle -q show
build/output/classes
```

然而， compileJava 任务很可能不是唯一需要知道类文件在哪里的任务。

Java 插件添加了 source sets 的概念 （见SourceSet） 来描述的源文件集的各个方面，其中一个方面是在编译的时候这些类文件应该被写到哪个地方。Java 插件将 compileJava 任务的 destinationDir 属性映射到源文件集的这一个方面。

我们可以通过这个源码集修改写入类文件的位置。

Example 21.8. Plugin convention object

build.gradle

```
apply plugin: 'java'

sourceSets.main.output.classesDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

执行 gradle -q show

```
> gradle -q show
build/output/classes
```

在上面的示例中，我们应用 Java 插件，除其他外，还做了下列操作：

- 添加了一个新的域对象类型： SourceSet
- 通过属性的默认（即常规）配置了 main 源码集
- 配置支持使用这些属性来执行工作的任务

所有这一切都发生在 `apply plugin: "java"` 这一步过程中。在上面例子中，我们在约定配置被执行之后，修改了类文件所需的位置。在上面的示例中可以注意到，compileJava.destinationDir 的值也被修改了，以反映出配置的修改。

考虑一下另一种消费类文件的任务的情况。如果这个任务使用 sourceSets.main.output.classesDir 的值来配置，那么修改了这个位置的值，无论它是什么时候被修改，将同时更新 compileJava 任务和这一个消费者任务。

这种配置对象的属性以在所有时间内（甚至当它更改的时候）反映另一个对象的任务的值的能力被称为“映射约定”。它可以令 Gradle 通过约定优于配置及合理的默认值来实现简洁的配置方式。而且，如果默认约定需要进行修改时，也不需要进行完全的重新配置。如果没有这一点，在上面的例子中，我们将不得不重新配置需要使用类文件的每个对象。

21.7. More on plugins 更多插件

这一章旨在作为对插件和 Gradle 及他们扮演的角色的导言。关于插件的内部运作的详细信息，请参阅Chapter 59. Writing Custom Plugins 编写自定义插件。

Chapter 22. Standard Gradle plugins 标准 Gradle 插件

Gradle 发布包包含了很多插件，如下：

22.1. Language plugins 语言类插件

这些插件添加各种语言可以被编译为在JVM中执行的支持。

Table 22.1. Language plugins

Plugin Id	Automatically applies	Works with	Description
`java`	`java-base`	-	Adds Java compilation, testing and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins. See also Chapter 7, *Java Quickstart*.
`groovy`	`java` , `groovy-base`	-	Adds support for building Groovy projects. See also Chapter 9, *Groovy Quickstart*.
`scala`	`java` , `scala-base`	-	Adds support for building Scala projects.
`antlr`	`java`	-	Adds support for generating parsers using Antlr.

22.2. Incubating language plugins 孵化中的语言插件

Table 22.2. Language plugins

Plugin Id	Automatically applies	Works with	Description
`assembler`	-	-	Adds native assembly language capabilities to a project.
`c`	-	-	Adds C source compilation capabilities to a project.
`cpp`	-	-	Adds C++ source compilation capabilities to a project.
`objective-c`	-	-	Adds Objective-C source compilation capabilities to a project.
`objective-cpp`	-	-	Adds Objective-C++ source compilation capabilities to a project.
`windows-resources`	-	-	Adds support for including Windows resources in native binaries.

22.3. Integration plugins 集成插件

这些插件提供一些集成各种运行技术。

Table 22.3. Integration plugins

Plugin Id	Automatically applies	Works with	Description
`application`	`java`	-	Adds tasks for running and bundling a Java project as a command-line application.
`ear`	-	`java`	Adds support for building J2EE applications.
`jetty`	`war`	-	Deploys your web application to a Jetty web container embedded in the build. See also Chapter 10, *Web Application Quickstart*.
`maven`	-	`java` , `war`	Adds support for publishing artifacts to Maven repositories.
`osgi`	`java-base`	`java`	Adds support for building OSGi bundles.
`war`	`java`	-	Adds support for assembling web application WAR files. See also Chapter 10, *Web Application Quickstart*.

22.4. Incubating integration plugins 孵化中的集成插件

这些插件提供一些集成各种运行技术。

Table 22.4. Incubating integration plugins

Plugin Id	Automatically applies	Works with	Description
`distribution`	-	-	Adds support for building ZIP and TAR distributions.
`java-library-distribution`	`java` , `distribution`	-	Adds support for building ZIP and TAR distributions for a Java library.
`ivy-publish`	-	`java` , `war`	This plugin provides a new DSL to support publishing artifacts to Ivy repositories, which improves on the existing DSL.
`maven-publish`	-	`java` , `war`	This plugin provides a new DSL to support publishing artifacts to Maven repositories, which improves on the existing DSL.

22.5. Software development plugins 软件开发插件

Table 22.5. Software development plugins

Plugin Id	Automatically applies	Works with	Description
`announce`	-	-	Publish messages to your favourite platforms, such as Twitter or Growl.
`build-announcements`	announce	-	Sends local announcements to your desktop about interesting events in the build lifecycle.
`checkstyle`	`java-base`	-	Performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.
`codenarc`	`groovy-base`	-	Performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.

`eclipse`	-	`java` , `groovy` , `scala`	Generates files that are used by Eclipse IDE, thus making it possible to import the project into Eclipse. See also Chapter 7, *Java Quickstart*.
`eclipse-wtp`	-	`ear` , `war`	Does the same as the eclipse plugin plus generates eclipse WTP (Web Tools Platform) configuration files. After importing to eclipse your war/ear projects should be configured to work with WTP. See also Chapter 7, *Java Quickstart*.
`findbugs`	`java-base`	-	Performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.
`idea`	-	`java`	Generates files that are used by Intellij IDEA IDE, thus making it possible to import the project into IDEA.
`jdepend`	`java-base`	-	Performs quality checks on your project's source files using JDepend and generates reports from these checks.
`pmd`	`java-base`	-	Performs quality checks on your project's Java source files using PMD and generates reports from these checks.
`project-report`	`reporting-base`	-	Generates reports containing useful information about your Gradle build.
`signing`	base	-	Adds the ability to digitally sign built files and artifacts.
`sonar`	-	java-base, java, jacoco	Provides integration with the Sonar code quality platform. Superceeded by the `sonar-runner` plugin.

22.6. Incubating software development plugins 孵化中的软件开发插件

Table 22.6. Software development plugins

Plugin Id	Automatically applies	Works with	Description
`build-dashboard`	reporting-base	-	Generates build dashboard report.
`build-init`	wrapper	-	Adds support for initializing a new Gradle build. Handles converting a Maven build to a Gradle build.
`cunit`	-	-	Adds support for running CUnit tests.
`jacoco`	reporting-base	java	Provides integration with the JaCoCo code coverage library for Java.
`sonar-runner`	-	java-base, java, jacoco	Provides integration with the Sonar code quality platform. Supersedes the `sonar` plugin.
`visual-studio`	-	native language plugins	Adds integration with Visual Studio.
`wrapper`	-	-	Adds a `Wrapper` task for generating Gradle wrapper files.
`java-gradle-plugin`	java		Assists with development of Gradle plugins by providing standard plugin build configuration and validation.

22.7. Base plugins 基本插件

这些插件形成基本构建块，用来提供给其他插件组装。它们可以被使用在你的构建文件，并列出的完整性。然而，要注意他们还没有考虑 Gradle 的公共API的一部分。因此，这些插件都不能列在用户指南文件中。你可以参考他们的API文档来了解他们。

Table 22.7. Base plugins

Plugin Id	Description
base	Adds the standard lifecycle tasks and configures reasonable defaults for the archive tasks: • adds build `ConfigurationName` tasks. Those tasks assemble the artifacts belonging to the specified configuration. • adds upload `ConfigurationName` tasks. Those tasks assemble and upload the artifacts belonging to the specified configuration. • configures reasonable default values for all archive tasks (e.g. tasks that inherit from `AbstractArchiveTask`). For example, the archive tasks are tasks of types: `Jar` , `Tar` , `Zip` . Specifically, `destinationDir` , `baseName` and `version` properties of the archive tasks are preconfigured with defaults. This is extremely useful because it drives consistency across projects; the consistency regarding naming conventions of archives and their location after the build completed.
java-base	Adds the source sets concept to the project. Does not add any particular source sets.
groovy-base	Adds the Groovy source sets concept to the project.
scala-base	Adds the Scala source sets concept to the project.
reporting-base	Adds some shared convention properties to the project, relating to report generation.

22.8. Third party plugins 第三方插件

可以从这里看到外部的插件

Chapter 23. The Java Plugin 关于 Java 插件

Java 插件添加 Java 编译和测试、捆绑的能力到项目中。这是许多其他 Gradle 插件的基础。

23.1. 用法

使用 Java 插件，添加如下脚本

Example 23.1. Using the Java plugin

build.gradle

```
apply plugin: 'java'
```

23.2. Source set

Java 插件引入了 source set 概念。 source set 就是一个源文件集合，用于编译和执行。这些源文件可能包含 Java 源文件和资源文件。其他插件添加这种包含 Groovy 和 Scala 源文件的能力。 source set 与编译 classpath 和运行时 classpath 关联。

source set 的一个用途是将源文件进行逻辑分组，这样可以描述他们的目的。例如，你可能使用 source set 来定义一个继承测试套件，或者你可能使用独立的 source set 来定义 API 和项目类的实现。

Java 插件定义了两个标准 source set ，称为 main 和 test。mian source set 包含您的生产源代码，并编译成一个 JAR 文件。test source set 包含您的测试源代码，这是使用 JUnit 和 TestNG 来编译和执行。这些可以是单元测试，集成测试，验收测试，或任何组合，对你非常有用。

23.3. Task（任务）

Java 插件添加了很多任务到你的项目中，如下：

Table 23.1. Java plugin - tasks

Task 名称	依赖于	类型	描述
`compileJava`	所有任务产生编译 classpath。这包含了 jar 任务给项目依赖，包含在编译配置中	`JavaCompile`	Compiles production Java source files 使用 javac 产生编译 Java 源文件
`processResources`	-	`Copy`	拷贝生产资源到生产类目下
`classes`	`compileJava` 任务和 `processResources` 任务。一些插件添加了额外的编译任务	`Task`	组装生产类目录
`compileTestJava`	`compile`，加上所有的任务产生测试编译 classpath	`JavaCompile`	使用 javac 来编译测试 Java 源文件
`processTestResources`	-	`Copy`	拷贝测试源文件到测试类目录
`testClasses`	`compileTestJava` 任务和 `processTestResources` 任务。一些插件添加了额外的测试编译任务	`Task`	组装生产类目录
`jar`	`compile`	`Jar`	组装 JAR 文件
`javadoc`	`compile`	`Javadoc`	使用 Javadoc 产生 API 文档给生产的 Java 源文件
`test`	`compile`，`compileTest`，加上所有的任务产生的测试运行时的classpath	`Test`	使用 JUnit 或 TestNG 执行单元测试
`uploadArchives`	任务产生在 `archives` 配置中的的构件,包含了 `jar` .	`Upload`	上传 `archives` 配置的构件，包含了 JAR 文件
`clean`	-	`Delete`	删除项目构建目录
`cleanTaskName`	-	`Delete`	删除特定任务的文件。 `cleanJar` 将会删除 `jar` 任务生产的 JAR 文件, `cleanTest` 将会删除 `test` 任务测试产生的结果

For each source set you add to the project, the Java plugin adds the following compilation tasks:

Appendix E. Existing IDE Support and how to cope without it 支持的 IDE 以及如何应对没有它

E.1. IntelliJ

Idea IntelliJ 可以很好的开发 Gradle ，提供了不错的插件。这个 IDE 同样支持 Gradle 的构建脚本。IntelliJ 允许您定义任何文件形式来被解释为一个 Groovy 脚本。在 Gradle ，您可以定义 build.gradle 和settings.gradle 这种模式。这就已经很有用。现在缺少的是路径的 Gradle 的二进制文件的 Gradle 类提供内容辅助。你可以添加 Gradle jar （你可以在你的发布包找到）到你的项目的类路径。它真的不属于那里，但如果你这样做，你将可以有一个出色的 IDE 支持开发 Gradle 脚本。当然，如果你使用其他库的构建脚本会进一步污染你的项目的类路径。

我们希望在未来的 *.gradle 文件在 IntelliJ 中能得到特殊待遇，你能为它们定义一个特定的路径。

E.2. Eclipse

在 Eclipse 有 Groovy 的插件。我们不知道它是什么样的状态，以及它是如何支持 Gradle。在本用户指南的下一版本我们希望能写更多关于这个。

E.3. Using Gradle without IDE support 无需IDE

我们能为您做些什么是让你输入东西就像抛出 new org.gradle.api.tasks.StopExecutionException() 那样，而仅仅需要输入抛出 new StopExecutionException() 来代替 。我们是通过添加 Gradle 脚本自动实现的。

Figure E.1. gradle-imports

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.cache.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
```

```
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.component.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.dsl.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.logging.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.jetty.*
import org.gradle.api.plugins.osgi.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.plugins.sonar.*
import org.gradle.api.plugins.sonar.model.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.buildinit.plugins.*
```

```
import org.gradle.buildinit.tasks.*
import org.gradle.external.javadoc.*
import org.gradle.ide.cdt.*
import org.gradle.ide.cdt.tasks.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.platform.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.plugins.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.coffeescript.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.java.tasks.*
import org.gradle.language.javascript.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.scala.*
import org.gradle.language.scala.plugins.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.scala.toolchain.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
```

```
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.test.*
import org.gradle.play.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
import org.gradle.plugins.javascript.envjs.http.simple.*
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
import org.gradle.plugins.javascript.rhino.worker.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.sonar.runner.*
import org.gradle.sonar.runner.plugins.*
import org.gradle.sonar.runner.tasks.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.util.*
```